Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Ferjančič

Medsistemska sprotna integracija izvorne kode

DIPLOMSKO DELO
UNIVERZITETNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Miha Mraz
MENTOR

Ljubljana, 2016

Tematika naloge:

Kandidat naj v svojem delu realizira sistem, ki omogoča hkratno sprotno integracijo programske kode na več platform z različnimi operacijskimi sistemi s poudarkom na prevajanih jezikih kot sta C in C++. Pri tem naj bo kandidat pozoren do željenih značilnosti sistema, ki naj bi bil javno dostopen in skalabilen glede na število projektov.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani izjavljam, da sem avtor dela, da slednje ne vsebuje materiala, ki bi ga kdorkoli predhodno že objavil ali oddal v obravnavo za pridobitev naziva na univerzi ali drugem visokošolskem zavodu, razen v primerih kjer so navedeni viri.
S svojim podpisom zagotavljam, da:

- sem delo izdelal samostojno pod mentorstvom Mihe Mraza,
- so elektronska oblika dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko in
- soglašam z javno objavo elektronske oblike dela v zbirki "Dela FRI".

— Klemen Ferjančič, Ljubljana, junij 2016.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Klemen Ferjančič

Medsistemska sprotna integracija izvorne kode

POVZETEK

Diplomsko delo opisuje načrtovanje in razvoj sistema, ki omogoča hkratno sprotno integracijo programske izvorne kode na več operacijskih sistemih s poudarkom na prevajanih jezikih kot sta C in C++. Glavni cilji sistema so javno dostopna storitev, kamor lahko vsak razvijalec doda svoj projekt izvorne kode, preprosta skalabilnost glede na število dodanih projektov in izolacija projektov v izogib konfliktov pri prevajanju in testiranju. Za doseganje skalabilnosti sistem izkorišča funkcionalnosti oblaka, ki omogoča avtomatiziran najem novih strežnikov glede na trenutno obremenjenost, za medsebojno izolacijo posameznih projektov pa izkorišča kontejnerske tehnologije tipa Docker in BSD Jails.

Ključne besede: sprotna integracija, jenkins, docker, jails, oblak, skalabilnost

University of Ljubljana
Faculty of Computer and Information Science

Klemen Ferjančič

Cross-platform continuous integration

ABSTRACT

This thesis describes planning and implementation of a system, which allows us to perform continuous integration of source code on several operating systems at once, with compiled languages such as C or C++ in mind. The main goals of the system are a publicly available service for developers, scalability as the quantity of added projects grows and isolation of the projects to avoid build and test conflicts. Scalability is achieved with cloud integration, which allows us to buy new servers automatically as the load increases. Isolation between projects is achieved with technologies such as Docker and BSD Jails.

Key words: continuous integration, cross-platform, jenkins, docker, jails, cloud, scalability

ZAHVALA

Zahvaljujem se družini in mentorju prof. dr. Mihi Mrazu za potrpežljivost in vzpodbudo pri izdelavi diplomskega dela.

— Klemen Ferjančič, Ljubljana, junij 2016.

KAZALO

1 Uvod

1.1 Motivacija

Zamislimo si, da smo kot razvijalec programske opreme del nekega odprtokodnega projekta. Kot večina popularnih projektov tudi naš repozitorij kode za verzioniranje uporablja git in gostuje na portalu GitHub. Recimo, da je projekt napisan v jeziku C++ in teče na vseh večjih operacijskih sistemih (Windows, Linux, Mac OS, FreeBSD...). Kako lahko zagotovimo, da se vsak prispevek kode še vedno zgradi na vseh platformah[1] in na vseh tudi deluje? Od vsakega razvijalca težko pričakujemo, da bo pri sebi poganjal vse štiri sisteme, zato potrebujemo storitev, ki bo našo kodo zgradila in stestirala na vseh platformah ob vsakem novem prispevku kode. Takšno storitev bi lahko nudil eden od razvijalcev na lastnih strežnikih, še najbolje pa bi bilo, da jo postavimo kar v oblak. Rešitev, ki jo opisujemo, se imenuje gostovana sprotna integracija (angl. *hosted continuous integration*), to je storitev sprotne integracije (v nadaljevanju CI), ki jo nekdo ponuja zastonj ali proti plačilu. Prednost teh rešitev je v tem, da nam ni potrebno vzdrževati

[1] Pojma operacijski sistem in platforma v tej nalogi pokrivata isti pojem in se bosta uporabljala izmenjaje.

lastne infrastrukture in CI sistemov, ampak se naša koda gradi in testira pri nekem zunanjem ponudniku. Vse kar moramo storiti z naše strani je storitvi povedati, s katerimi ukazi naj gradi kodo, nazaj pa dobimo rezultate v obliki dnevniških zapisov (logov) ali statusnih ikon (npr. zelena ikona pomeni da se je projekt zgradil, rdeča da se ni). Večina teh storitev je zastonjskih za odprtokodne projekte, hkrati pa se znajo tudi integrirati z GitHubom in podobnimi portali, kar je vidno iz prej omenjenih statusnih ikon, ki jih lahko dodamo na spletno stran projekta in tako vizualno predstavimo v kakšnem stanju je trenutno naša koda. Ena od pomanjkljivosti teh storitev je, da večinoma podpirajo le en operacijski sistem, redko dva [1] ali več, kar predstavlja težavo za razvijalce medplatformskih projektov. Tega problema nimamo pri interpretiranih jezikih kot je Java, saj se ta na vseh platformah izvaja v istem virtualnem stroju, medtem ko moramo prevajane jezike graditi na vsaki platformi posebej, včasih tudi z različnimi prevajalniki. Cilj te diplomske naloge je implementacija "hosted CI" storitve, ki lahko projekte gradi in testira na več različnih platformah hkrati.

1.2 Opis problema

Opisan problem lahko strnemo v nekaj naslednjih točk:

- Imamo projekt v git repozitoriju.
- Projekt je napisan v prevajanem jeziku, npr. C ali C++.
- Projekt deluje na več operacijskih sistemih.
- Želimo izvajati sprotno integracijo na vseh podprtih platformah hkrati.
- Rešitev želimo izpostaviti kot javno storitev, kamor lahko poljuben razvijalec doda svoj projekt.
- Rešitev mora biti skalabilna in mora znati rasti skupaj s številom dodanih projektov.
- V kolikor se na enem izmed sistemov (virtualki, strežniku, sužnju[2]) hkrati gradi in testira več projektov, naj bodo med seboj izolirani, da ne prihaja do konfliktov ali zlonamernih napadov.

[2] S pojmom suženj v nalogi mislimo virtualni ali fizični strežnik, ki gradi in testira projekte.

- Rešitev naj v celoti deluje v oblaku, zato da nam ni treba postavljati lastne infrastrukture.

1.3 Pregled dela

Najprej bomo v poglavju 2 razložili pomen sprotne integracije na več platformah in katere vrste problemov želimo na tem področju rešiti. Poseben poudarek bomo namenili izpostavitvi sistema sprotne integracije kot javne storitve, skalabilnosti sistema in izolaciji projektov. Nato bomo v poglavju 3 zasnovali načrt rešitve, ki bo zadostil zahtevani poslovni logiki in reševal uvodno zadane cilje. V poglavju 4 bomo podrobno predstavili implementacijo rešitve. S pomočjo izsekov kode in skript bodo razložene pomembnejše komponente sistema. Predstavljena bo tudi implementacija spletnega vmesnika z zaslonskimi slikami.

2 Sprotna integracija

2.1 Opis pojma sprotne integracije

Sprotna integracija je eden izmed dvanajstih principov ekstremnega programiranja (angl. *extreme programming*) in spodbuja prispevanje kode v centralen repozitorij večkrat na dan, kjer se koda avtomatično gradi in testira. Kent Beck, avtor knjige Extreme Programming Explained [2], je sprotno integracijo definiral s stavkom "Continuous integration—Integrate and build the system many times a day, every time a task is completed.". V originalnem priporočilu se omenjajo časovni okviri integracije (npr. vsakih nekaj ur ali enkrat na dan), dandanes pa se v praksi integracija sproži kar ob vsakem prispevku kode (angl. *commit*) v repozitorij.

Kot dobro prakso jo izpostavljajo tudi agilne metode, ki z njo dosežejo, da [3]

- minimiziramo čas in trud vsakega integracijskega cikla in
- da smo v vsakem trenutku sposobni izdati novo delujočo verzijo.

Osnovna ideja metode je zgodnje odkrivanje in popravljanje napak. S tem se predvsem pri velikih projektih izognemo končnemu združevanju večjih delov kode in mr-

zličnemu odpravljanju napak, ko se bliža dan nove izdaje.

Velikokrat se skupaj s CI uporablja tudi tako imenovan TDD (angl. *Test Driven Development*). TDD zahteva, da se hkrati ali celo pred dejansko kodo napiše tudi teste.

Eden od problemov zvezne integracije je ta, da je lahko začetna vzpostavitev sistema zamudna. Prav tako lahko CI porabi precej virov in računskih zmogljivosti, za kar je potrebna dovolj zmogljiva infrastruktura. Zato se zdi, da se nam predvsem pri manjših in osebnih projektih CI ne izplača. Vendar so raziskave na realnih projektih pokazale, da nam uporaba CI zmanjša t.i. "check-in" časa od 40% za manjše projekte in še več za velike [4]. To je čas, ki ni namenjen programiranju in predstavlja "overhead" pri razvoju projekta.

Rešitev za problem začetne vzpostavitve CI in vzdrževanja infrastrukture so javno dostopne storitve, ki namesto nas gostujejo CI programsko opremo in vzdržujejo infrastrukturo. Najpogosteje se s temi storitvami projekti integrirajo tako, da se v repozitorij kode doda konfiguracijska datoteka, ki vsebuje odvisnosti projekta, kako ga zgraditi, ukaze za testiranje in podobno. Periodično ali ob vsakem doprinosu kode nam potem zunanja storitev avtomatično projekt zgradi in stestira. Po končanem ciklu lahko vidimo izpis in dnevnik gradnje projekta, na voljo pa so nam tudi statusne ikone, ki prikazujejo "zdravje" projekta in jih lahko dodamo na lastno spletno stran.

2.2 Medsistemska sprotna integracija

Osnovni problem razvoja programske opreme je tveganje in eno izmed teh tveganj je število napak v kodi [2]. Ko imamo opravka s projekti, ki so namenjeni večim platformam (angl. *cross-platform*[1]), postane CI bolj zapleten, saj moramo sedaj projekt graditi in testirati na več platformah hkrati. Vsaka sprememba kode lahko namreč deluje na eni izmed platform, na drugih pa povzroči napako.

Zaradi uporabe več različnih platform je tveganje zato še toliko večje, saj se mora aplikacija izvajati na enak način v različnih okoljih. Sistem za CI mora zato imeti podporo za več platform in sicer na način mojster-suženj (angl. *master-slave*), kar pomeni, da lahko s centralnega strežnika vodimo vse gradnje na različnih platformah istočasno.

Ne samo da projekte gradimo na več platformah hkrati, na vsaki platformi lahko uporabimo tudi več različnih prevajalnikov, lahko pa gre celo za različne procesorske

[1] Projekt, ki je namenjen delovanju na več platformah oziroma operacijskih sistemih.

arhitekture. Grajenje projekta z več prevajalniki je predvsem pri medplatformskih projektih dobra praksa, saj so nekateri prevajalniki pametnejši od drugih, opozorijo na napake ali izpišejo dodatna opozorila, ki so razvijalcu lahko v pomoč pri odkrivanju napak. Zanimiv je eksperiment Sevana Janiyana, ki je z orodjem pkgsrc poskušal graditi 15.000 različnih projektov na več različnih *NIX platformah in samo z grajenjem (brez zagona programov) kode našel napake v projektih in celo operacijskih sistemih [5].

2.3 Skalabilnost sistema

V razdelku 1.2 smo kot eno izmed problematik izpostavili skalabilnost sistema. Ko se naši storitvi pridruži vse več projektov lahko kmalu ugotovimo, da naše začetne kapacitete strežnikov niso več dovolj in potrebujemo nove. Če je naš sistem zasnovan pravilno s tem ni večjih težav, saj so projekti medsebojno neodvisni in imamo takorekoč idealno horizontalno skalabilnost. Če storitev gostujemo na lastni infrastrukturi to pomeni nakup nove strojne opreme, še raje pa se poslužimo oblačnih ponudnikov, ki nam nov strežnik lahko vzpostavijo v nekaj minutah s klicem na oblačni REST API. Tako na novo pridobljen strežnik povežemo s centralnim CI sistemom, ki ga upravlja kot sužnja in na njem izvaja grajenje in testiranje kode. Skaliranje lahko izvajamo ročno kot sistemski administrator, še bolje pa bi bilo, da je sistem tako pameten, da ob povečani obremenitvi kar sam najame nove strežnike in o tem dogodku administratorja kvečjemu obvesti. Prav takšno avtomatizirano skalabilnost smo tudi implementirali v tej nalogi.

2.4 Javna izpostavitev storitve

Dokler CI sistem uporabljamo za lastne projekte nimamo večjih varnostnih problemov in konfliktov med projekti. Konfliktov je lahko več vrst, eden najbolj preprostih pa je ta, da nek test projekta A zasede določena TCP vrata, hkrati pa se zažene še test projekta B, ki prav tako želi poslušati na istih vratih. Ker so vrata zasedena, test projekta B pade, čeprav v resnici ni šlo za napako v projektu ampak za zunanji vpliv. Do tega konflikta pride zato, ker se testiranja posameznih projektov lahko izvajajo paralelno [6]. Ker so to naši lastni projekti in ker imamo poln nadzor na CI sistemom se v večini primerov lahko takšnim konfliktom izognemo na primer tako, da zahtevamo zaganjanje testov enega za drugim. Če izpostavimo našo CI storitev javnosti, se nenadoma na istem strežniku znajde več različnih projektov, ki eden za drugega ne vedo in kmalu lahko pride do prej

omenjenih konfliktov. Tu je še varnostni vidik, saj lahko kdorkoli na našo storitev odda zlonameren projekt, ki bi lahko škodil samemu sistemu ali oviral ostale legitimne projekte. Za grajenje kode razvijalec načeloma potrebuje proste roke pri izbiri ukazov, ki naj se na naših strežnikih izvedejo, saj so lahko postopki za gradnjo predvsem večjih projektov zelo zapleteni in se torej ne moremo omejiti samo na vnaprej omejen nabor (varnih) ukazov. Prav tako moramo razvijalcem omogočiti namestitev vseh paketov programske opreme, od katerih je njihov projekt odvisen, kar pomeni, da dovoljujemo poganjanje ukazov kot administrator (na *NIX sistemih kot uporabnik z imenom root). Gradnjo projektov in izvajanje njihovih testov moramo torej nujno izolirati med sabo. Preprosta rešitev je ta, da imamo za vsak projekt ločen virtualni strežnik, kar pa je zelo potratno in cenovno težko izvedljivo. Namesto tega namenimo vsakemu projektu lasten kontejner, to je neke vrste zelo lahek virtualni strežnik, ki za izolacijo virov izkorišča funkcionalnosti jedra operacijskega sistema. Na enem virtualnem strežniku imamo tako lahko na desetine (ali več, odvisno od zmogljivosti strežnika) kontejnerjev, programi, ki se v njih izvajajo, pa se drug drugega ne zavedajo. S kontejnerji dosežemo izolacijo projektov in hkrati boljši izkoristek virov.

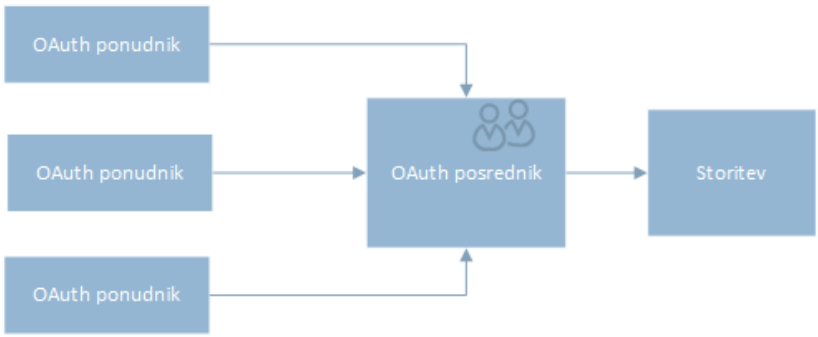
3 Načrt arhitekture

3.1 Spletni vmesnik

Spletni vmesnik je javno dostopna komponenta, do katere dostopajo uporabniki. V našem primeru bodo to predvidoma razvijalci programske opreme. Tu lahko dodajajo in urejajo svoje projekte. Hkrati spletni vmesnik prikazuje tudi trenutno stanje sistema, kamor sodijo podatki kot so kateri projekti se trenutno gradijo, kakšno je njihovo stanje, koliko sužnjev je trenutno del sistema in kakšna je njihova zasedenost.

3.1.1 Prijava

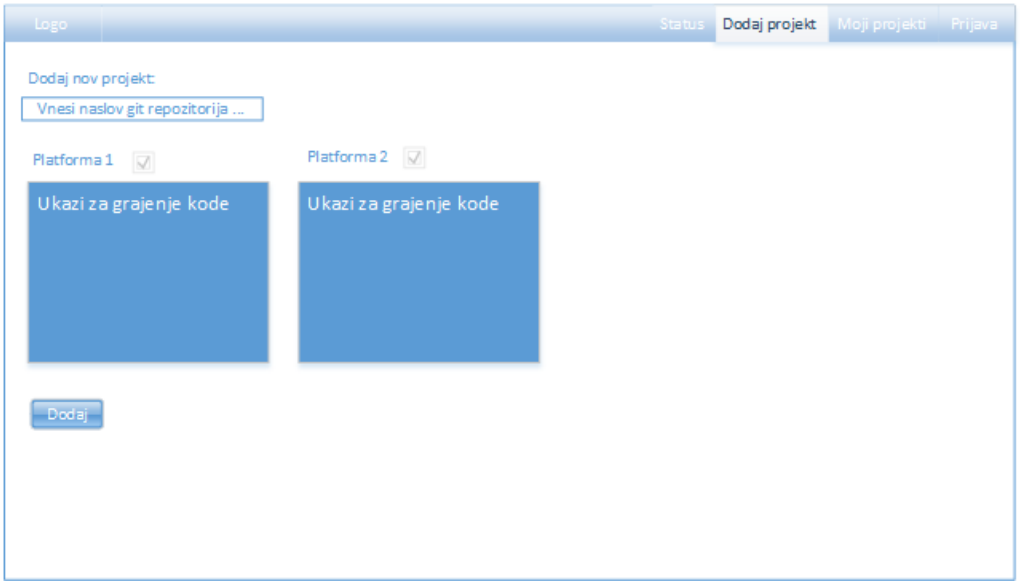
Vsak uporabnik potrebuje račun, s katerim se prijavi v našo storitev. Ker ima dandanes velika večina razvijalcev že račun na eni ali več trenutno popularnih storitev za gostovanje kode (GitHub, Bitbucket), se lahko na našo storitev prijavijo kar z obstoječim računom prek protokola OAuth. V našem sistemu zato uporabimo OAuth posrednika, ki nam omogoča avtentikacijo iz večih obstoječih OAuth ponudnikov. Razmerje med OAuth ponudniki, posrednikom in našim sistemom je predstavjeno na sliki 3.1.



Slika 3.1: Arhitektura avtentikacije.

3.1.2 Dodajanje projekta

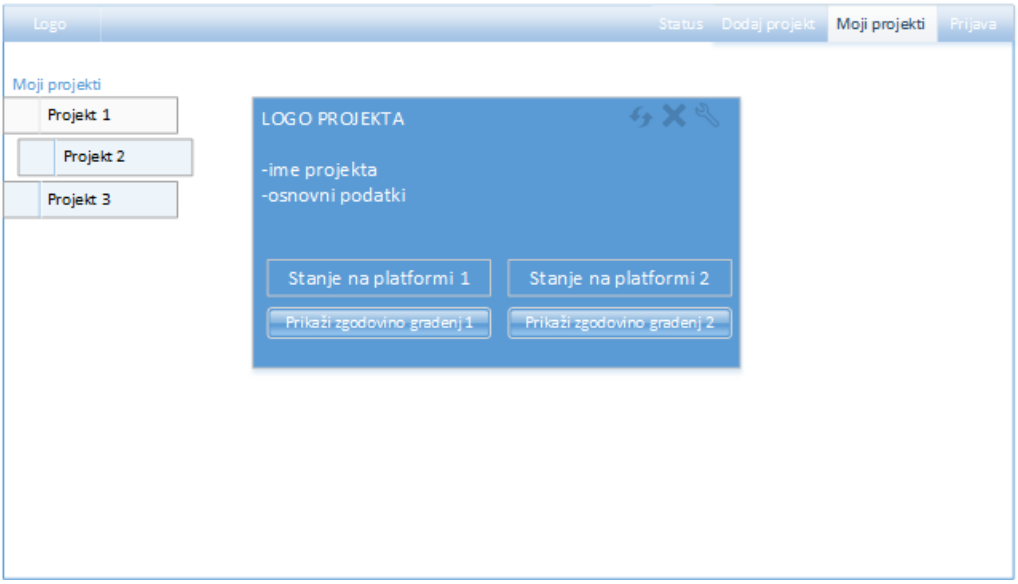
Po uspešni prijavi v sistem je eno izmed najbolj pomembnih opravil dodajanje novih projektov. Uporabnik mora definirati povezavo do repozitorija kode svojega projekta in napisati ukaze, ki kodo zgradijo, za vsako platformo posebej. Uporabnik lahko poljubno izbira na katerih platformah naj se gradi njegov projekt, s tem da željene platforme odkljuka. Model zavihka "Dodaj projekt" vidimo na sliki 3.2. Kljub temu, da je obrazec zelo preprost, dodajanje novega projekta v ozadju sproži precej kompleksnih procesov, kar si bomo ogledali v razdelku 3.3.



Slika 3.2: Obrazec za dodajanje novega projekta.

3.1.3 Moji projekti

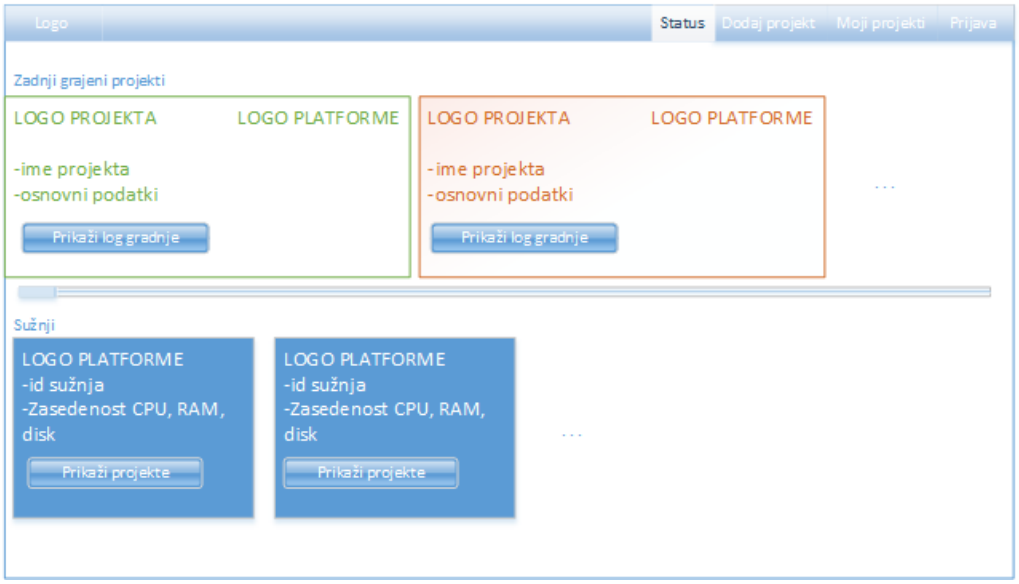
V zavihku "Moji projekti", ki je prikazan na sliki 3.3, si uporabnik lahko ogleda in spreminja že dodane projekte. Ob izbiri enega izmed projektov prikažemo njegove podrobnosti kot so ime projekta, logotip, naslov git repozitorija ter ukaze za grajenje. Za vsako platformo, na kateri se projekt gradi, prikažemo barvno statusno ikono, ki predstavlja t.i. "zdravje" projekta. S klikom na gumb lahko odpremo zgodovino gradenj v obliki seznama. Za vsako gradnjo vidimo status in dnevniški zapis, iz katerega razberemo, zakaj je pri neki gradnji prišlo do napake.



Slika 3.3: Model zavihka "Moji projekti".

3.1.4 Status sistema

V zavihku "Status", ki je prikazan na sliki 3.4, lahko vidimo, kaj se trenutno dogaja s celotnim sistemom in vse razpoložljive vire. Na vrhu strani se v živo posodabljajo nazadnje grajeni projekti, v spodnjem delu pa prikažemo vse razpoložljive sužnje. Za vsakega sužnja v obliki grafa predstavimo zasedenost virov (procesorski čas, pomnilnik in disk), logotip platforme in seznam projektov, ki se gradijo na tem sužnju.

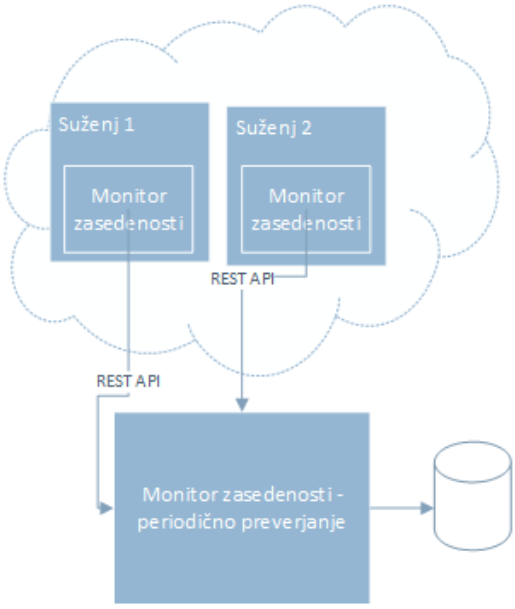


Slika 3.4: Model zavihka "Status".

3.2 Monitoring in upravljanje oblaka

Komponenta za upravljanje oblaka skrbi za to, da sistemu za sprotno integracijo nikoli ne zmanjka sužnjev za grajenje kode. Opravlja naslednje naloge:

- ob prvem zagonu sistema v oblaku najame po en virtualni strežnik (suženj) za vsako podprto platformo,
- ob zagonu novega sužnja nanj namesti in zažene vso potrebno programsko opremo,
- na sužnja namesti komponento za monitoring zasedenosti procesorja, pomnilnika in diska,
- periodično od vsakega sužnja zahteva informacijo o zasedenosti virov in odgovor shrani v podatkovno bazo,
- izračuna povprečno zasedenost vseh sužnjev in v primeru prevelike zasedenosti v oblaku najame nove sužnje,
- ko je dodan nov projekt najde prostega sužnja za vsako platformo in na njem ustvari nov kontejner.



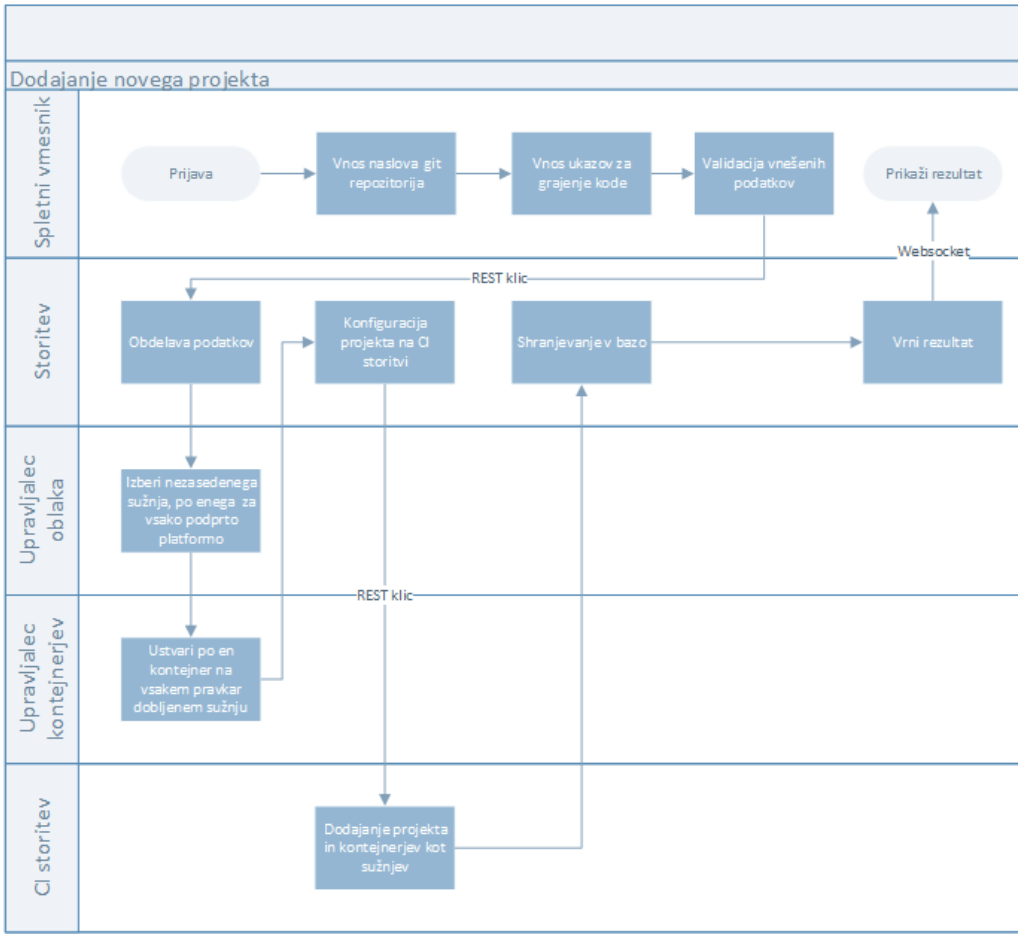
Slika 3.5: Model delovanja monitoringa.

Komponenta za monitoring je preprost program, ki se namesti na sužnja in prek REST vmesnika vrne zasedenost procesorja, pomnilnika in diska. Način delovanja je predstavljen na sliki 3.5.

3.3 Jedro sistema

Jedro sistema skrbi za izvajanje vse poslovne logike in integracijo komponent. Glavne funkcionalnosti jedra so:

- izpostavlja REST vmesnik, ki ga uporablja spletna stran za svoje delovanje,
- pridobiva podatke in izvaja poslovno logiko za potrebe spletne strani,
- ob dodajanju novega projekta prek komponente za upravljanje oblaka sproži kreiranje novih kontejnerjev,
- nov projekt doda v CI storitev,
- od CI storitve pridobiva podatke o stanju posameznih projektov in njihovo zgodovino gradnje.



Slika 3.6: Diagram poteka za dodajanje novega projekta.

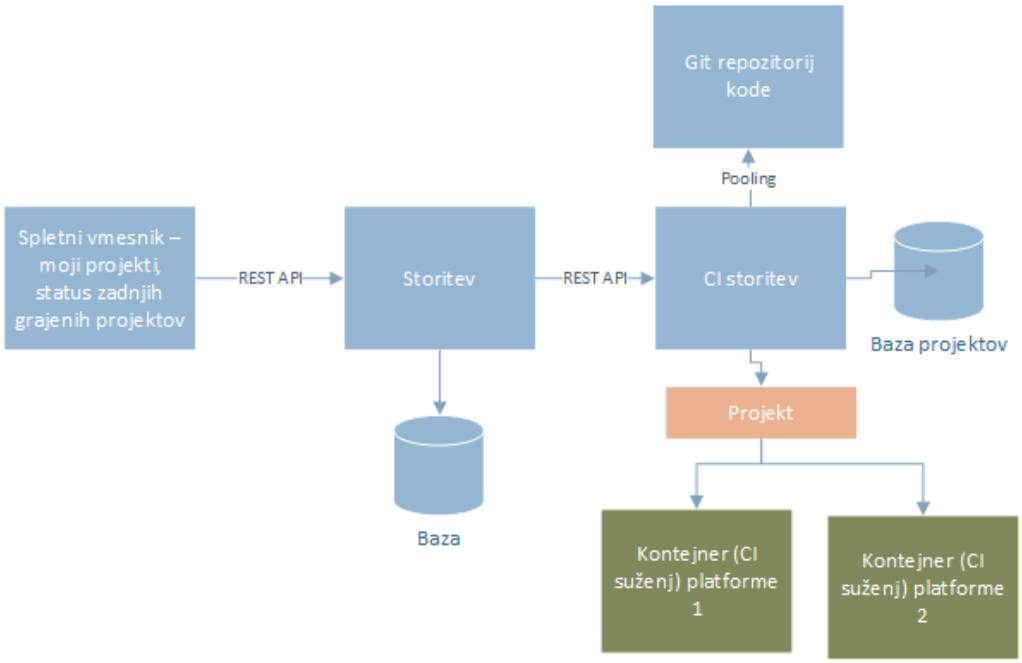
V razdelku 3.1.2 smo omenili, da dodajanje novega projekta v ozadju sproži precej kompleksnih procesov. Ker je to ena izmed bolj pomembnih operacij, smo na sliki 3.6 prikazali diagram poteka.

Uporabnik se prijavi na spletno stran in izpolni obrazec za dodajanje novega projekta. Vnešeni podatki se najprej validirajo (preveri se prazna polja, pravilnost povezave do repozitorija...) in prek REST vmesnika pošljejo v jedro sistema. Jedro podatke shrani v bazo in od komponente za upravljanje z oblakom zahteva prostega sužnja za vsako podprto platformo. Nato od iste komponente za vsakega vrnjenega sužnja zahteva kreiranje novega kontejnerja. Jedro nato izvede dva klica na CI storitev. Prvi v CI storitev doda kontejnerje kot sužnje. Drugi klic doda nov projekt, skonfiguriran na način, da se uporabnikov projekt gradi v lastnih kontejnerjih. Rezultati klicev se zapišejo v bazo in

na koncu jedro na spletni vmesnik vrne rezultat.

3.4 CI storitev

CI storitev je ločena komponenta, ki izvaja sprotno integracijo projektov. Zaradi velike kompleksnosti in zahtevnosti razvoja takšne programske opreme bomo v naš sistem vključili že obstoječo rešitev in te komponente ne bomo razvijali sami.

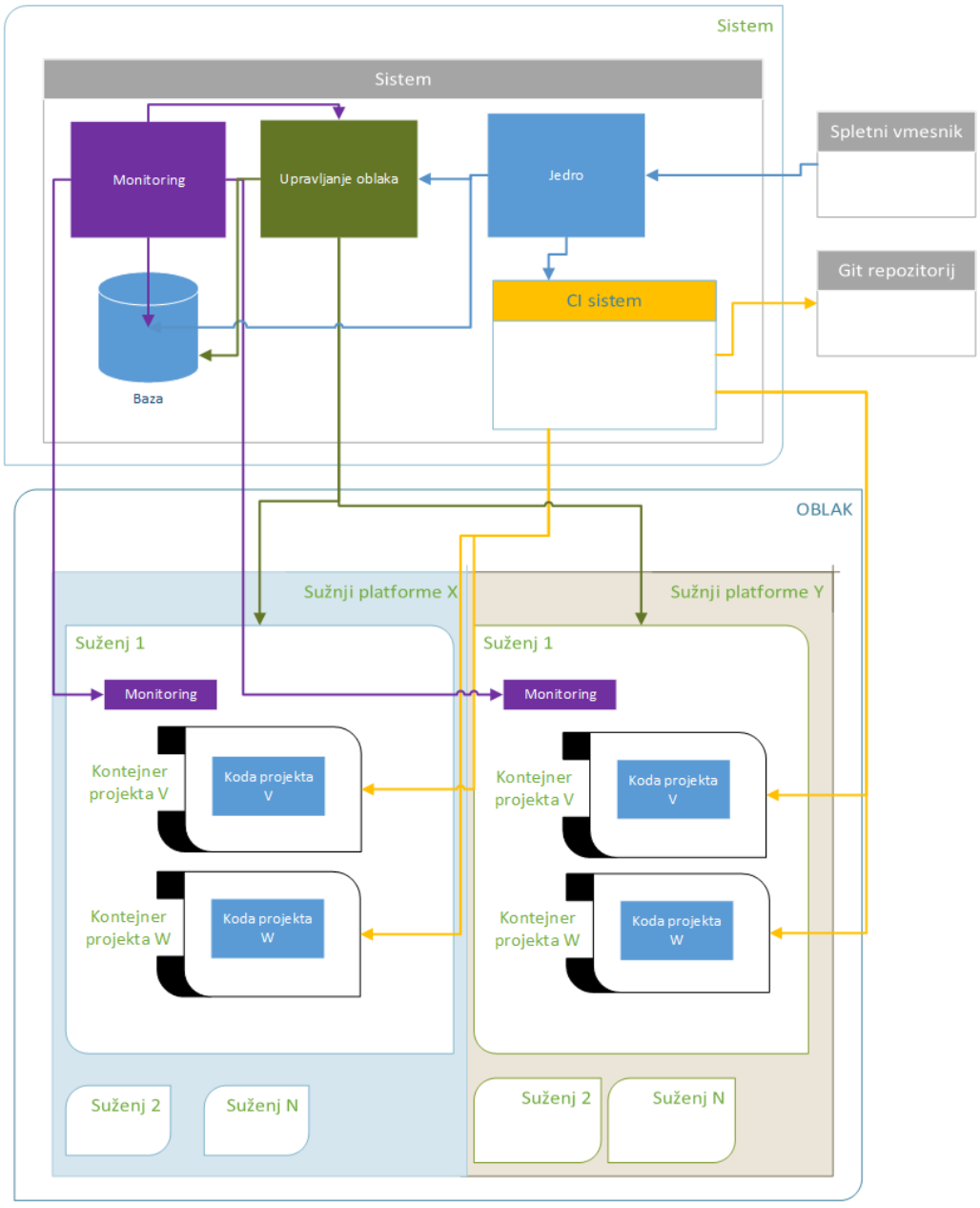


Slika 3.7: Integracija CI storitve z ostalimi komponentami.

Izbrana CI storitev mora navzven izpostavljati REST vmesnik, zato da lahko naš sistem z njo upravlja. Prek REST vmesnika na CI storitev dodajamo nove projekte in njihove sužnje. Po tem koraku CI storitev načeloma deluje sama od sebe, naš sistem pa le še poizveduje po statusu in zgodovini grajenja posameznih projektov za potrebe spletnega vmesnika. Nove gradnje projektov se sprožijo takrat, ko CI storitev zazna spremembe v repozitoriju kode. Za detekcijo sprememb v kodi se uporablja periodično preverjanje (angl. *pooling*) prispevkov kode v repozitorij. Integracija in delovanje CI storitve je predstavljena na sliki 3.7.

3.5 Načrt celotnega sistema

Vse prej opisane komponente združimo v visokonivojski načrt končnega sistema. Slika 3.8 prikazuje vse komponente in njihovo interakcijo med seboj, bazami podatkov in zunanjimi storitvami.



Slika 3.8: Visokonivojski načrt vseh komponent končnega sistema.

4 Implementacija rešitve

4.1 Uporabljene tehnologije

Jenkins Jenkins je odprtokodno orodje za sprotno integracijo napisan v Javi. Razvil ga je Kohsuke Kawaguchi leta 2004 pri podjetju Sun. Sprva se je orodje imenovalo Hudson, leta 2010 pa se je zaradi spora z Oraclom večina razvijalcev odcepila in projekt preimenovala v Jenkins. Jenkins je vodilno orodje na področju sprotne integracije in je leta 2010 obladovalo 70% trga [6]. Je zelo razširljiv, saj zanj obstaja precej vtičnikov, ki razširjajo njegove funkcionalnosti. Lahko ga upravljamo prek spletnega ali REST vmesnika. Deluje lahko v *master-slave* konfiguraciji, pri čemer uporablja sužnje za grajenje kode in poganjanje testov. Jenkins izvaja poglavitno nalogo našega sistema, to je sprotno integracijo, zato je arhitektura sistema zasnovana v tem smislu, da se čim bolj prilagodi načinu delovanja Jenkinsa.

jclouds Apache JClouds je odprtokodna Javanska knjižnica, ki nam abstraktira uporabo oblačnih ponudnikov. Namesto klicanja specifičnih APIjev nekega oblačnega ponudnika uporabimo generične klice, ki delujejo za vse bolj znane ponudnike.

Za oblačnega ponudnika smo izbrali Amazon Web Services EC2, pri kateremu najemamo virtualke (sužnje) za grajenje kode. Za ponudnika smo se odločili zaradi dobre podpore v knjižnici JClouds, raznolike ponudbe operacijskih sistemov in cene.

Vsa poslovna logika sistema je napisana v Javi EE 7 in teče na odprtokodnem aplikacijskem strežniku WildFly 10. Za to rešitev smo se odločili, ker sta Jenkins in JClouds napisana v Javi in zaradi lastnih izkušenj s to tehnologijo.

Za implementacijo spletnega vmesnika smo uporabili ogrodje AngularJS. Orodje bazira na JavaScriptu in nam omogoča popolno razklopljenost spletnega vmesnika s poslovno logiko. Spletna stran je popolnoma neodvisna od implementacije poslovne logike in z njo komunicira prek REST vmesnika.

KeyCloak je avtentikacijski strežnik, ki skrbi za uporabniške račune, avtentikacijo in avtorizacijo uporabnikov. Uporabljamo ga kot OAuth posrednika, kar nam omogoča, da se uporabniki prijavijo z že obstoječimi računi drugih storitev.

KumuluzEE je ogrodje za razvoj Java EE mikrostoritev. Uporabljamo ga za razvoj monitoring komponente, ki na sužnjih izpostavlja njihovo zasedenost prek REST vmesnika. S tem je velikost namestitve veliko manjša, saj bi v nasprotnem primeru monitoring morali namestiti na aplikacijski strežnik, kar je zelo potratno.

4.2 Podprte platforme

Število podprtih platform (operacijskih sistemov) neposredno vpliva na to, kako uporabna je naša storitev za razvijalce *cross-platform* programov, zato stremimo k temu, da bi bila ta številka čim višja, vendar ne za vsako ceno. Podpora platforme mora biti tehnično izvedljiva in ekonomsko upravičena. Izmed štirih najbolj razširjenih namiznih in strežniških operacijskih sistemov v tem trenutku smo se na koncu odločili za dva in sicer za Linux in FreeBSD.

4.2.1 Linux

Za Linux distribucijo smo izbrali Ubuntu 14.04 LTS, najnovejšo stabilno različico z dolgoročno podporo. Razen velike razširjenosti te distribucije in lastnega poznavanja ni poglavitnega razloga, zakaj smo izbrali ravno Ubuntu. V splošnem bi lahko uporabili katerokoli moderno Linux distribucijo. Tehnologija na Linux sistemih, ki nas najbolj zanima, je Docker in vse večje distribucije Linuxa ga podpirajo, zato je sama izbira bolj stvar osebnih preferenc.

Docker

Kontejnerje na nivoju operacijskega sistema si lahko predstavljamo kot zelo lahke virtualke, ki namesto virtualizacije strojnih komponent za izolacijo procesov izkoriščajo funkcionalnosti jedra operacijskega sistema. Kontejnerji so star koncept. UNIX sistemi imajo že desetletja orodje chroot, ki predstavlja preprosto obliko izolacije na nivoju datotečnega sistema. Od leta 1998 ima operacijski sistem FreeBSD orodje Jails, ki razširja funkcionalnosti chroot. Solaris Zones je podoben koncept ponudil leta 2001. Nato je Google začel razvoj CGroups (izolacija na nivoju virov CPU, pomnilnika in diska). Linux Containers (LXC) projekt, ki se je začel leta 2008, je povezal vse prej omenjene tehnologije v celovito kontejnersko rešitev. Končno je Docker leta 2013 dokončal sestavljanko in kontejnerje predstavil širši množici uporabnikov, predvsem v obliki uporabniku prijaznega vmesnika in centralnega registra kontejnerjev z imenom Docker Hub [7]. Omogoča nam, da lahko le z nekaj ukazi vzpostavimo nov kontejner, v katerem teče naša aplikacija. Kontejnerje lahko zaganjamo, ustavljamo, zbrišemo, se povežemo na njihov terminal... Kontejner ustvarimo iz opisne datoteke z imenom Dockerfile, v kateri opišemo njegovo strukturo.

Docker za svoje delovanje izkorišča Linux jedro, zato primarno deluje le na Linuxu. Delna implementacija Dockerja obstaja tudi na operacijskem sistemu FreeBSD [8],vendar ne podpira vseh ukazov in za delovanje zahteva FreeBSD 11.0-CURRENT, to je najnovejša še neizdana verzija in je zato ne bomo našli pri oblačnih ponudnikih. Na platformi FreeBSD bomo torej morali uporabiti drugo kontejnersko tehnologijo.

4.2.2 FreeBSD

FreeBSD je prosto dostopen odprtokodni operacijski sistem, ki izvira iz Unixa. Je tretji najbolj uporabljan operacijski sistem na strežnikih za Linuxom in Windowsi [9]. V nasprotju z Linux distribucijami, kjer se jedro in uporabniška oprema razvijata posebej, se FreeBSD razvija kot celota. Za naš sistem najbolj zanimiva tehnologija so FreeBSD Jails (sl. *zapori*), ki je FreeBSDjeva implementacija kontejnerjev.

Jails

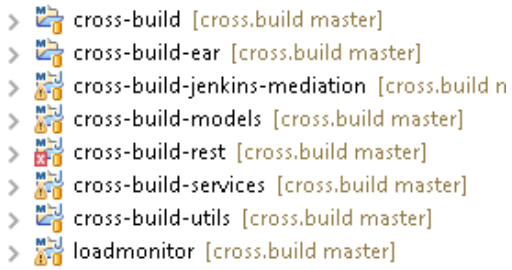
Jails je kontejnerska tehnologija, ki predstavlja varnostni mehanizem in virtualizacijo na nivoju operacijskega sistema. Ponuja enake funkcionalnosti kot Linux ekvivalent. Tudi zapori bazirajo na chroot mehanizmu. Tako kot Docker izpostavlja kontejnerje z uporabniku prijaznim vmesnikom ima tudi FreeBSD orodja, ki nam olajšajo delo z zapori. Najbolj znani so Ezjail, Qjail, locage in Jetpack. Pred začetkom implementacije smo stestirali vsa omenjena orodja in se na koncu odločili za Ezjail. Glavna prednost orodja Ezjail je ta, da podpira predloge, ki so podoben mehanizem kot Dockerfile in nam omogoča, da nove zapore baziramo na obstoječi predlogi.

4.2.3 Windows, Mac OS

V kolikor bi naš sistem lahko podpiral tudi Windows in Mac OS operacijska sistema, bi s tem pokrili skoraj ves trg namiznih in strežniških operacijskih sistemov. Na žalost ti dve platformi v času pisanja te diplomske naloge nista ustrezali skalabilni arhitekturi sistema. Noben izmed njiju trenutno ne podpira kontejnerskih tehnologij, zato bi kot sužnje za Jenkins morali uporabljati celotne virtualke, kar je potratna in cenovno neizvedljiva rešitev. Windows Server 2016 naj bi imel podporo za Docker [10], vendar ta verzija uradno še ni izdana, datum izida pa še ni znan. Mac OS ima še to pomanjkljivost, da ga ne moremo najeti pri nobenem večjem oblačnem ponudniku. Prav tako Mac OS ni dovoljeno nameščati v obliki virtualnih strežnikov, ki ne tečejo na Apple strojni opremi.

4.3 Organizacija kode projekta

Koda diplomske naloge je prosto dostopna na naslovu `https://github.com/cen1/cross.build` in je izdana pod BSD licenco, ki jo najdemo v datoteki z imenom LICENSE. Pro-



Slika 4.1: Organizacija Maven projektov v okolju Eclipse.

jekt je organiziran kot standarden Maven projekt, katerega struktura je prikazana na sliki 4.1. Vsebuje naslednje komponente:

- cross-build-ear, arhiv za namestitev na aplikacijski strežnik,
- cross-build-jenkins-mediation, je mediacijski projekt za upravljanje orodja Jenkins prek REST vmesnika
- cross-build-models, vsebuje JPA entitete za upravljanje s podatkovno bazo,
- cross-build-rest, implementira REST API za komunikacijo s spletnim vmensikom,
- cross-build-services, vsebuje poslovno logiko zalednih storitev,
- cross-build-utils, komponenta s splošno uporabnimi razredi,
- loadmonitor, ki vsebuje mikrostoritev za monitoring,

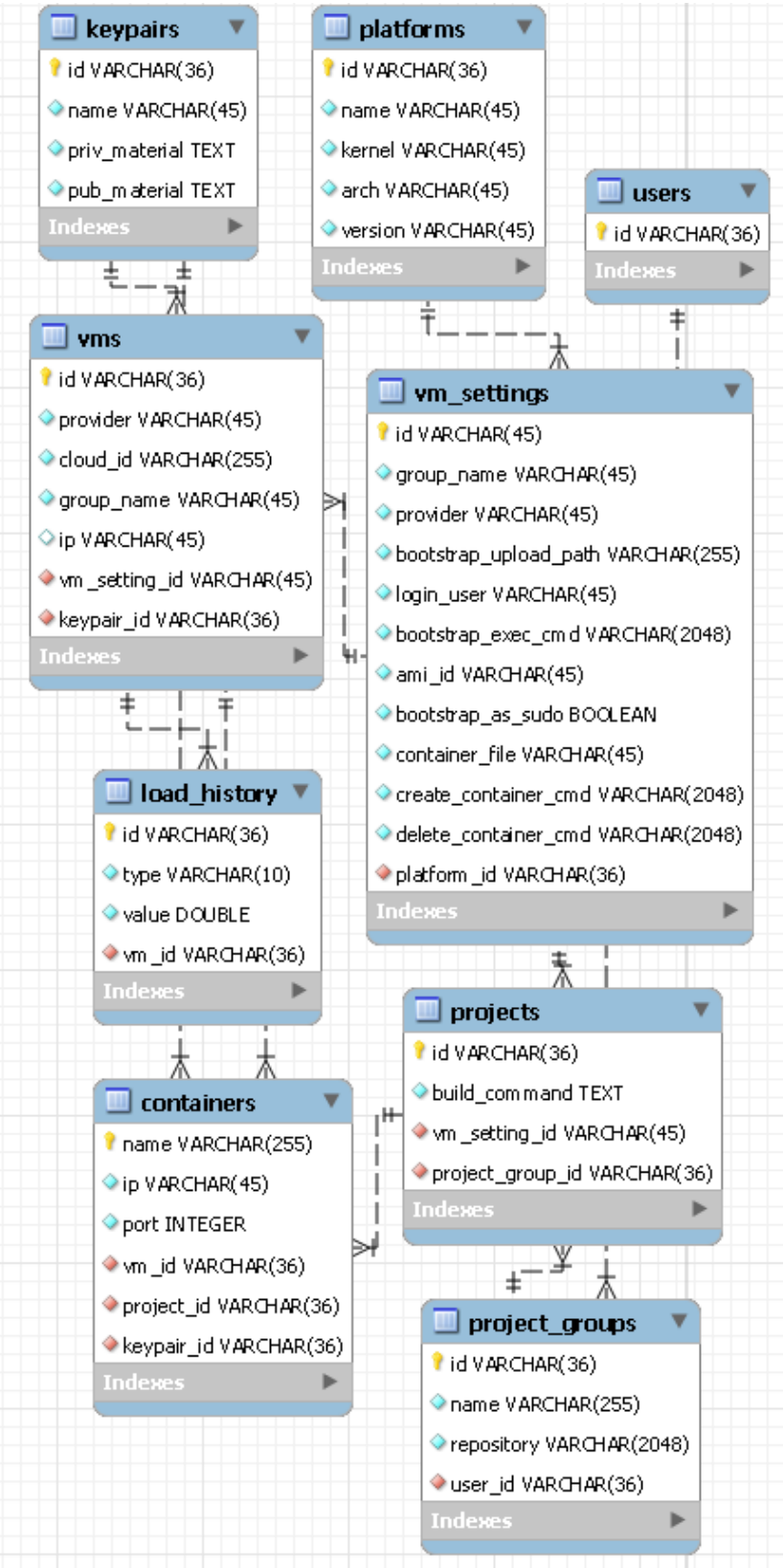
Maven projekte najdemo v mapi z imenom core. Poleg tega imamo še mapo z imenom web, ki vsebuje spletni vmesnik v ogrodju AngularJS in pa mapo deployment, ki vsebuje skripte za lažjo namestitev sistema.

4.4 Jedro

Jedro je skupek projektov, ki implementirajo glavnino poslovne logike in skrbijo za integracijo sistemov.

4.4.1 Podatkovni model

Model baze je relativno majhen. Predstavljen je na sliki 4.2. Za migracijo baze uporabljamo orodje Liquibase, kot bazo pa PostgreSQL. Tabele so v Javi izpostavljene kot



Slika 4.2: Model baze v orodju MySQL Workbench.

objekti z modulom JPA, nahajajo pa se v mapi *core/models*. Baza vsebuje naslednje tabele:

- **vms**, vsebuje vse virtualke, ki jih najemamo pri oblačnem ponudniku; ob najemu virtualke si shranimo njen IP, ime skupine (v našem primeru kar AMI ID ponudnika) in ID virtualke, ki ga nastavi oblačni ponudnik; kot tuja ključa dodamo še entiteti vm_settings in keypairs,
- **vm_settings**, vsebuje vse potrebne podatke za najem in upravljanje nekega operacijskega sistema pri določenem oblačnem ponudniku; vnosi v tej tabeli so izpostavljeni na spletnem vmesniku kot seznam vseh platform, ki so na voljo za uporabo; tabela vsebuje ime privzetega uporabnika za SSH prijavo, poti do direktorijev za nalaganje skript, ime datoteke za opis kontejnerja ter ukaze za zagon in izbris kontejnerjev; kot tuj ključ se doda tabela platforms,
- **keypairs**, vsebuje SSH ključe za povezavo na najete virtualke prek knjižnice JClouds,
- **platforms**, opisuje tip operacijskega sistema; vsebuje ime operacijskega sistema, tip jedra, CPU arhitekturo in verzijo,
- **load_history**, shranjuje celotno zgodovino zasedenosti virtualk; en vnos pomeni zasedenost določenega vira neke virtualke ob nekem času; za preverjanje zasedenosti virtualk se nad to tabelo izvajajo izračuni povprečij,
- **project_groups**, predstavlja abstraktno entiteto, ki skupaj združuje več projektov; gre za projekte, ki izhajajo iz istega repozitorija kode in se gradijo na različnih platformah; pripada uporabniku storitve, zato ima tudi tuji ključ tabele users,
- **projects**, gre za projekt, ki se gradi na eni izmed platform; vsebuje ukaz za grajenje kode, prek tujega ključa pripada projektni skupini (project_groups) in neki platformi (vm_settings),
- **containers**, predstavlja kontejner na virtualki; vsebuje polja ip in port, da se lahko nanj povežemo prek SSH protokola iz orodja Jenkins; za potrebe prijave prek protokola SSH se prek tujega ključa povezuje na tabelo keypairs; pripadnost virtualki se doseže s tujim ključem vm_id, pripadnost projektu pa s tujim ključem project_id,

- **users**, predstavlja uporabnika našega sistema; vsebuje le polje id, ki je identifikator uporabnika v bazi KeyCloak; uporabnik se doda v tabelo ob prvem avtenticiranem REST klicu na našo storitev.

4.4.2 REST vmesnik

REST vmesnik predstavlja vir podatkov za spletno stran. Nahaja se v mapi *core/rest* oziroma v Maven paketu *cross-build-rest*. Izpostavljeni viri se nahajajo v paketu *build.cross.rest.v1.resources* in razširjajo razreda CrudResource ali CrudUserResource. Prvi je namenjen splošnim entitetam, drugi pa entitetam, ki so vezana na uporabnika (npr. projektne skupine). CrudResource razred implementira naslednje metode:

- GET getList(), vrne seznam entitet in podpira tako imenovane "query" parametre, s katerimi povemo številko začetnega elementa, število željenih elementov, v glavi odgovora pa dobimo število vseh entitet, kar lahko uporabimo za paginacijo,
- GET get(String id), vrne entiteto z zahtevanim IDjem,
- POST create(Entity entity), ustvari novo entiteto in jo shrani v bazo,
- PUT update(String id, Entity entity), posodobi obstoječo entiteto z novimi podatki,
- GET count(), vrne število entitet,
- metoda validateEntity(String id) preveri, če entiteta z IDjem obstaja v bazi.

CrudUserResource deluje zelo podobno kot CrudResource, razlika pa je le v tem, da se v vse SQL klice doda še ID uporabnika, ki je sprožil klic. Metoda za pridobitev seznama tako vrne le entitete, ki pripadajo temu uporabniku in ne vseh. S tem ko REST viri razširjajo ta dva razreda nam je prihranjena implementacija standardnih GET, POST in PUT metod za vsak vir posebej. Ena izmed bolj zanimivih metod je metoda za dodajanje novih projektnih skupin, zato si poglejmo njeno implementacijo v izpisu kode 4.1 iz razreda ProjectGroupResource.

```
@EJB
private ContainerManagerSbLocal cmng;

@EJB
private NodeMediationSbLocal jenkinsNodes;

@EJB
private ProjectMediationSbLocal jenkinsProjects;

@POST
@Transactional(rollbackOn = {Exception.class,
    RuntimeException.class})
public Response addNew(ProjectGroup projectGroup) throws
    ApiException {

  User user = em.find(User.class,
      sc.getUserPrincipal().getName());
  projectGroup.setUser(user);
  em.persist(projectGroup);

  for (Project project : projectGroup.getProjects()) {
    project.setId(null);
        project.setProjectGroup(projectGroup);
        try {
      project = cmng.createNewContainer(project);
        } catch (ServiceException e) {
          throw new ApiException("Bad request", new
              ApiError(400, e.getMessage()));
        }
        em.persist(project);
        jenkinsNodes.createNode(project.getContainer());
```

```
            jenkinsProjects.addProject(project);
            jenkinsProjects.configureProject(project);
        }

        return Response.ok(projectGroup).build();
    }
```

Izpis kode 4.1: REST metoda za dodajanje nove projektne skupine.

Metoda se kliče s HTTP zahtevkom *POST http://naslov/v1/users/userId/projectgroups*, ki ima v telesu zahtevka entiteto ProjectGroup v JSON obliki. Projektni skupini nastavimo trenutno prijavljenega uporabnika, nato pa za vsak projekt iz seznama naredimo sledeče:

1. ustvarimo nov kontejner s pomočjo EJB zrna ContainerManagerSbLocal; več o implementaciji tega zrna bo povedano v razdelku 4.6.4,
2. pravkar ustvarjen kontejner dodamo v Jenkins kot sužnja s pomočjo mediacijskega zrna NodeMediationSbLocal,
3. ustvari nov projekt na Jenkinsu in ga pravilno skonfigurira s pomočjo mediacijskega zrna ProjectMediationSbLocal; več o mediacijskih zrnih bo povedano v razdelku 4.4.3.

4.4.3 Jenkins mediacija

Komponenta za Jenkins mediacijo izvaja REST klice na Jenkins s katerimi izvajamo konfiguracijo Jenkinsa, dodajanje novih sužnjev, dodajanje novih projektov in pridobivanje podatkov o grajenju projektov (angl. *build jobs*). Klici se izvajajo s pomočjo REST klienta, odgovori pa se v večini primerov vrnejo na spletni vmesnik v nespremenjeni obliki. Kljub temu spletni vmesnik nikoli direktno ne komunicira z Jenkins REST vmesnikom, ampak vedno le z našim lastnim. Želimo namreč imeti popoln nadzor nad tem kje se projekti gradijo in kdo lahko zahteva informacije o gradnjah. Jenkins se zato vzpostavi s samo enim skrbniškim računom, s katerim se avtenticira REST klient mediacije, registracija novih računov in sam dostop do Jenkinsa pa je navzven zaprt. REST operacije, ki jih izvaja mediacija nad Jenkinsom so sledeče: dodajanje SSH ključev za povezavo na kontejnerje, dodajanje kontejnerjev kot sužnjev, dodajanje projektov, izbris projektov

in kontejnerjev, pridobivanje podatkov o gradnjah. Jenkins na žalost ne podpira čisto vseh operacij preko standardnega REST vmesnika, zato smo določene operacije morali izvesti na način, da smo operacijo najprej izvedli prek Jenkins spletnega vmesnika, posneli HTTP promet, nato pa enak zahtevek implementirali na klientu. V večini primerov gre za posnemanje kompleksnih HTTP zahtev s HTML obrazci. Takšen pristop je bil potreben pri skoraj vseh klicih. V primeru klica za konfiguriranje projekta je REST klic deloval le na pol. Shranil se je na primer repozitorij, ime projekta in ukazi za grajenje, manjkali pa so podatki o tem, na katerem sužnju naj se projekt gradi in na koliko časa naj preverja git repozitorij. Tudi takšne na pol delujoče REST klice smo zato morali implementirati s pošiljanjem HTML obrazcev.

Zaradi omenjenih težav je koda mediacije precej "grda", zato si poglejmo le krajši odsek inicializacije in primer klica za dodajanje SSH ključev. Vsi razredi mediacije razširjajo razred MediationCommonSb, v katerem se izvede inicializacija predstavljena v izseku kode 4.2. Iz sistemskih spremenljivk se prebere naslov, uporabniško ime in geslo. Iz tega se zgradi niz, ki se uporabi v HTTP glavi Authorization, kar služi za osnovno avtentikacijo (angl. *basic authentication*).

```
@PostConstruct
protected void init() {
  jenkinsApi=System.getProperty("JENKINS_API");
  jenkinsUser=System.getProperty("JENKINS_USER");
  jenkinsPassword=System.getProperty("JENKINS_PASSWORD");

  String usernameAndPassword = jenkinsUser + ":" +
      jenkinsPassword;
  authorizationHeaderValue = "Basic " +
      java.util.Base64.getEncoder()
    .encodeToString(usernameAndPassword.getBytes());
}
```

Izpis kode 4.2: Inicializacija parametrov v razredu MediationCommonSb.

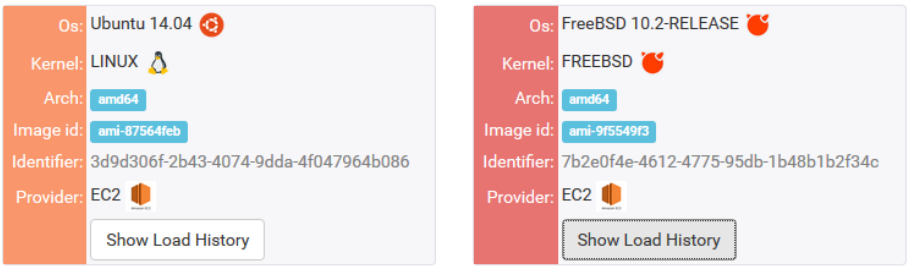
```
Response response =
    wt.path("/credential-store/domain/_/createCredentials")
  .header("Authorization", authorizationHeaderValue)
  .post(Entity.entity(form,
  MediaType.APPLICATION_FORM_URLENCODED));
```

Izpis kode 4.3: Primer klica z REST klientom na Jenkins.

Primer klica je prikazan v izpisu kode 4.3. Nad razredom WebTarget specificiramo pot, dodamo avtentikacijsko HTTP glavo in telo zahtevka v obliki obrazca. Nazaj dobimo objekt tipa Response, ki ga po potrebi spremenimo, v večini primerov pa kar posredujemo naprej v nespremenjeni obliki.

4.5 Spletni vmesnik

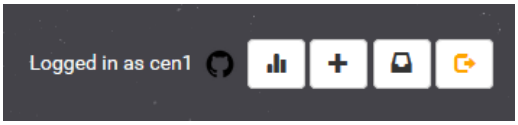
Spletni vmesnik služi uporabnikom storitve za dodajanje in upravljanje s projekti. Demonstracija storitve je dostopna na spletnem naslovu `http://cross.build`. Prva stran služi opisu storitve in informacijam o projektu. Brez prijave si lahko ogledamo še statusno stran, ki smo jo zasnovali v razdelku 3.1.4. Končni rezultat statusne strani je prikazan na sliki 4.3. Namero o prikazovanju nazadnje zgrajenih projektov smo opustili, ker Jenkins ne izpostavlja REST vmesnika, ki bi nam vrnil podatke o nazadnje zgrajenih projektih. Tako na statusni strani prikazujemo le trenutno najete virtualke in njihovo stanje. Pogled lahko še razširimo, da se prikažejo grafi zasedenosti virov, kar je predstavljeno na sliki 4.4.



Slika 4.3: Statusna stran spletnega vmesnika.

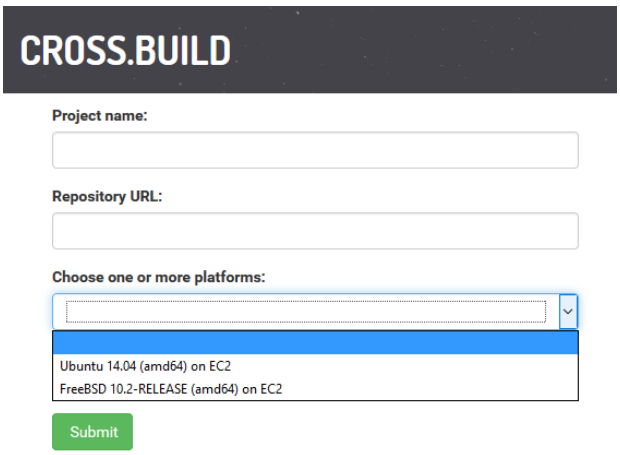
Slika 4.4: Grafi zasedenosti virov posamezne virtualke.

Po prijavi z računom GitHub se nam v orodni vrstici, prikazani na sliki 4.5, pojavijo dodatni gumbi s povezavami do GitHub repozitorija, statusne strani, strani za dodajanje projektov in strani s seznamom že dodanih projektov.

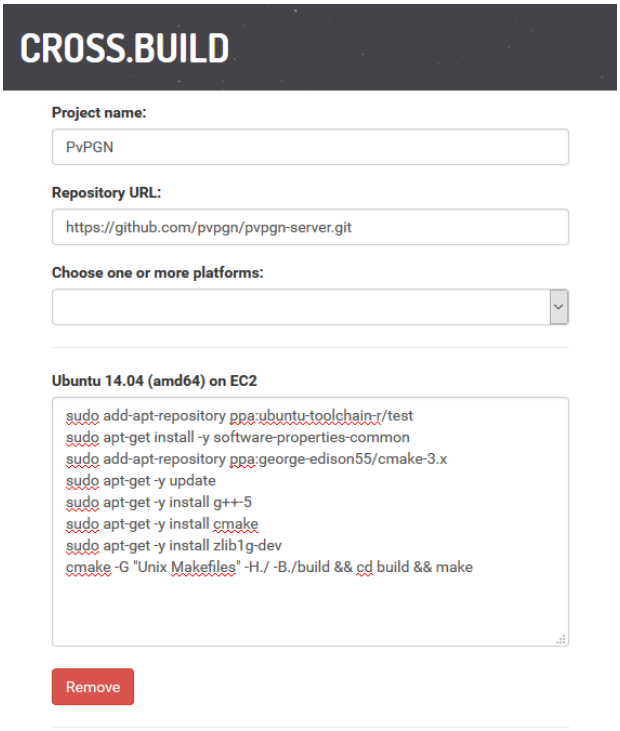


Slika 4.5: Orodna vrstica po prijavi v sistem.

S klikom na gumb plus v orodni vrstici se nam odpre obrazec za dodajanje novega projekta, ki je prikazan na sliki 4.6. Tu vnesemo naslov do našega git repozitorija, projektu damo ime in izberemo platforme, na katerih želimo graditi. Za vsako platformo specificiramo še ukaze za grajenje kode. Primer izpolnjenega obrazca je na sliki 4.7.



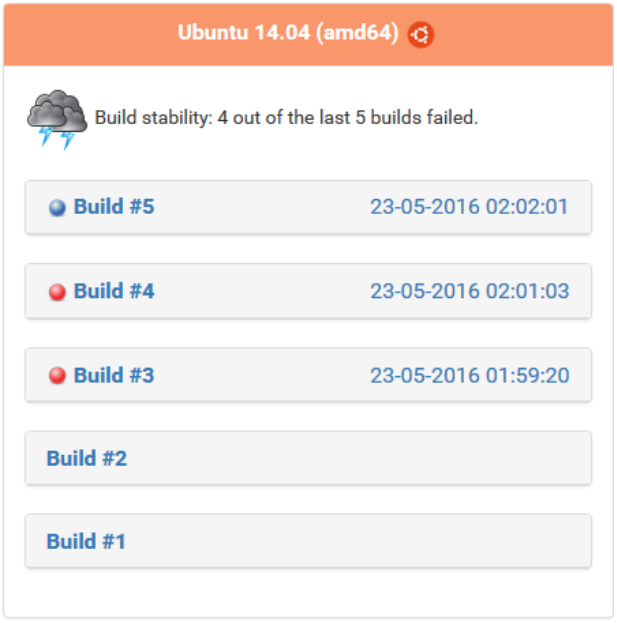
Slika 4.6: Obrazec za dodajanje novega projekta.



Slika 4.7: Primer izpolnjenega obrazca za dodajanje novega projekta.

Ko shranimo nov projekt nas vmesnik preusmeri na stran “Moji projekti”, kjer vidimo vse naše projekte in njihovo zgodovino. Primer za Ubuntu je prikazan na sliki 4.8, primer za FreeBSD pa na sliki 4.9. Na vrhu je prikazano splošno stanje projekta s simboli za

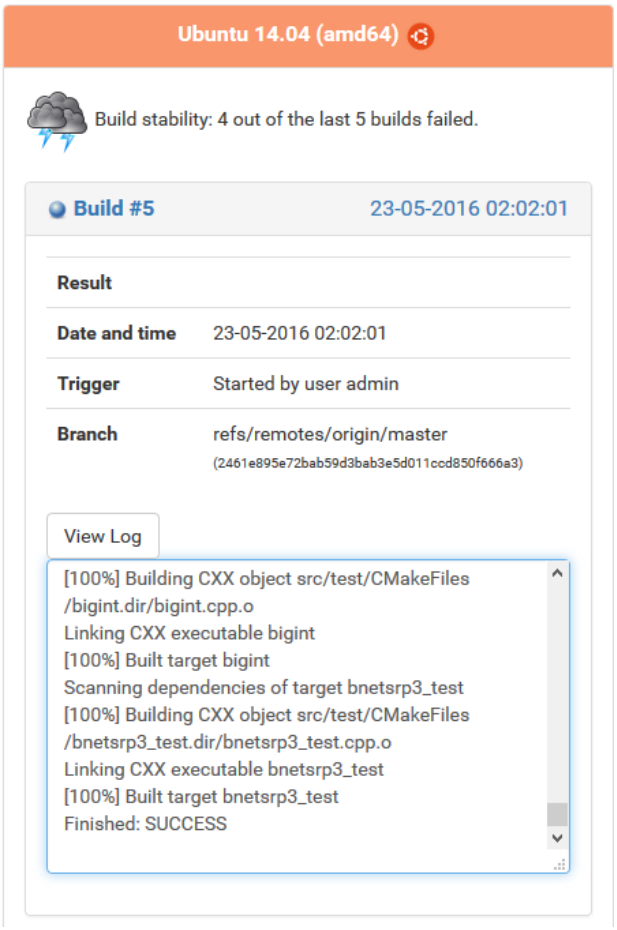
vreme, kot jih uporablja tudi Jenkins. Večkrat kot gradnja kode uspe, bolj "sončno" je stanje projekta na določeni platformi. Prikazan je tudi seznam vseh gradenj skupaj z zaporedno številko, datumom in statusno ikono, ki predstavlja uspeh ali neuspeh.



Slika 4.8: Zgodovina gradenj projekta na platformi Ubuntu.

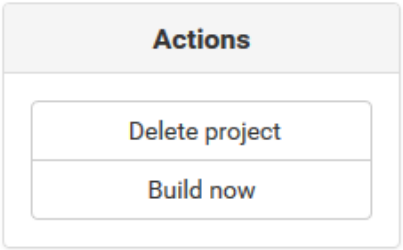
Slika 4.9: Zgodovina gradenj projekta na platformi FreeBSD.

Vsako gradnjo lahko tudi odpremo in takrat se prikaže več podrobnosti, vključno z dnevniškimi zapisi iz konzole, kar je prikazano na sliki 4.10.



Slika 4.10: Za vsako gradnjo lahko odpremo tudi dnevniški zapis, kjer je viden potek gradnje.

Za vsak projekt imamo na voljo še dve akciji in sicer izbris ter ročno proženje gradnje, kar je razvidno iz slike 4.11.



Slika 4.11: Vsak projekt lahko zbrišemo ali pa ročno sprožimo gradnjo.

4.6 Upravljanje oblaka

Komponenta za upravljanje oblaka skrbi za najemanje in upravljanje virtualnih strežnikov, ki jih naš sistem uporabi za grajenje in testiranje projektov. Komponenta skrbi tudi za kreiranje kontejnerjev in njihovo pravilno namestitev. Ko so zmogljivosti presežene, komponenta avtomatsko najame nove strežnike. Podatke o zmogljivosti pridobi od komponente za monitoring, ki jo namesti na vsak strežnik.

4.6.1 Najem nove virtualke

Za najemanje novih virtualk uporabljamo knjižnico JClouds, kot ponudnika pa uporabljamo Amazon AWS, kot smo omenili že v poglavju 4.1. Organizacija kode je sledeča:

- paket *build.cross.services.cloud.management* vsebuje naslednje razrede:
  - VmManagerSb, EJB zrno brez stanja, implementira glavnino poslovne logike za upravljanje z virtualkami,
  - VmManagerSbLocal, lokalni vmesnik, ki definira javno dostopne metode razreda VmManagerSb,
  - VmInitializer, enkratni EJB števec, ki se izvede ob prvem zagonu sistema in najame začetne virtualke,
  - CryptographySb, EJB zrno brez stanja, komponenta za ekstrakcijo javnega ključa iz privatnega ključa
  - CryptographySbLocal, lokalni vmesnik, ki definira javno dostopne metode razreda CryptographySb.

Začnimo z razlago razreda VmManagerSb in si oglejmo njegove lastnosti na izpisu kode 4.4.

```
@PersistenceContext(unitName = "crossbuild")
private EntityManager em;

private Logger logger =
    Logger.getLogger(VmManagerSb.class.getSimpleName());

@EJB
private CryptographySbLocal crypto;

private String CLOUD_PROVIDER;
private String CLOUD_IDENTITY;
private String CLOUD_CREDENTIALS;
private String CLOUD_REGION;
private String CLOUD_INSTANCE_TYPE;

private ComputeService compute;
```

Izpis kode 4.4: Lastnosti razreda VmManagerSb.

Najprej definiramo EntityManager za upravljanje z bazo podatkov in instanco razreda Logger za izpisovanje logov. Sledi pet tekstovnih spremenljivk, ki se definirajo kot lastnosti na aplikacijskem strežniku WildFly in predstavljajo konfiguracijo oblaka:

- *CLOUD_PROVIDER* pove, katerega oblačnega ponudnika uporabljamo (v našem primeru je to *aws-ec2*),
- *CLOUD_IDENTITY* pridobimo iz AWS konzole; je kot nekakšno uporabniško ime,
- *CLOUD_CREDENTIALS* prav tako pridobimo iz AWS konzole, predstavlja geslo; *CLOUD_IDENTITY* in *CLOUD_CREDENTIALS* dovoljujeta avtentikacijo na AWS REST API,
- *CLOUD_REGION* pove, kateri podatkovni center po svetu želimo uporabljati; odločili smo se za *eu-central-1* v Frankfurtu,
- *CLOUD_INSTANCE_TYPE* predstavlja zmogljivost virtualk, ki jih bomo najemali, npr. *t2.micro*.

Spremenljivke dodamo v konfiguracijo aplikacijskega strežnika kot sistemske lastnosti. Primer brez zaupnih vrednosti je podan na izpisu kode 4.5.

```
<system-properties>
   <property name="CLOUD_PROVIDER" value="aws-ec2"/>
   <property name="CLOUD_IDENTITY" value=""/>
   <property name="CLOUD_CREDENTIALS" value=""/>
   <property name="CLOUD_REGION" value="eu-central-1"/>
   <property name="CLOUD_INSTANCE_TYPE" value="t2.micro"/>
</system-properties>
```

Izpis kode 4.5: Konfiguracija aplikacijskega strežnika v datoteki standalone_full.xml.

Zadnja lastnost je razred ComputeService, ki je vstopna točka za vso interakcijo z oblačnim APIjem. Konfiguracijske spremenljivke in ComputeService razred se instancirajo v metodi init(), ki se sproži po konstrukciji razreda VmManagerSb. Metoda je prikazana na izpisu kode 4.6.

```
@PostConstruct
private void init() {

CLOUD_PROVIDER = System.getProperty("CLOUD_PROVIDER");
CLOUD_IDENTITY = System.getProperty("CLOUD_IDENTITY");
CLOUD_CREDENTIALS = System.getProperty("CLOUD_CREDENTIALS");
CLOUD_REGION = System.getProperty("CLOUD_REGION");
CLOUD_INSTANCE_TYPE =
   System.getProperty("CLOUD_INSTANCE_TYPE");

compute = initComputeService(CLOUD_PROVIDER,
   CLOUD_IDENTITY, CLOUD_CREDENTIALS);
}
```

Izpis kode 4.6: Init metoda razreda VmManagerSb.

Po inicializaciji razreda lahko končno začenjamo najemati nove virtualke. To storimo s pomočjo metod prepareTemplate() in createVm(). S pomočjo izpisa kode 4.7 si najprej oglejmo metodo prepareTemplate().

```
private TemplateBuilder prepareTemplate(VmSettings
    vmSettings) throws ServiceException, IOException {
 TemplateBuilder templateBuilder =
     compute.templateBuilder();
 templateBuilder.imageId(CLOUD_REGION + "/" +
     vmSettings.getAmiId());
 templateBuilder.locationId(CLOUD_REGION);
 templateBuilder.hardwareId(CLOUD_INSTANCE_TYPE);

 EC2TemplateOptions o =
     EC2TemplateOptions.Builder.keyPair("crossbuild")
  .overrideLoginCredentials(
  getLoginForCommandExecution(vmSettings))
  .securityGroups("crossbuild");
 templateBuilder.options(o);

 return templateBuilder;
}
```

Izpis kode 4.7: Metoda prepareTemplate() razreda VmManagerSb.

Metoda vrača razred TemplateBuilder, iz katerega lahko zgradimo predlogo virtualke, sprejme pa razred VmSettings, ki je naš lasten razred, ki opisuje tip virtualke, ki ga želimo najeti. Najbolj pomembni lastnosti razreda VmSettings sta *amiId* in *user*. Prva določi identifikator predloge na oblaku EC2. Vsaka verzija operacijskega sistema, ki jo lahko najamemo, ima takšen identifikator, ki je odvisen tudi od regije, v kateri najemamo. V našem primeru uporabljamo operacijska sistema Ubuntu 14.04 in FreeBSD 10 z identifikatorjema ami-87564feb in ami-9f5549f3. Identifikator za Ubuntu je kar uradna različica, ki jo ponuja Amazon, identifikator za FreeBSD pa je privatna različica, ki jo ponuja eden izmed razvijalcev FreeBSD. Najdemo jo na naslovu https://aws.amazon.com/marketplace/pp/B00KSS55FY/. Ob prvi uporabi predloge

FreeBSD moramo ročno potrditi, da se strinjamo s pogoji uporabe te predloge. Temu koraku smo se želeli izogniti, vendar AWS nima nobenega APIja, preko katerega bi lahko sprožili strinjanje avtomatično iz našega sistema. Lastnost user pove, kako je ime privzetemu uporabniku virtualke. Za predlogo Ubuntu je to "ubuntu", za predlogo FreeBSD 10 pa "ec2-user". Ime uporabnika moramo vedeti zato, da se lahko v virtualko kasneje prijavimo. V metodi s pomočjo razreda VmSettings in konfiguracijskih spremenljivk zgradimo predlogo. Dodamo še ime ključa za avtentikacijo, ki se v našem primeru imenuje "crossbuild". Ključ se kreira na EC2 ob prvem zagonu sistema (več o tem v poglavju 4.6.2). Na koncu podamo še ime varnostne skupine, ki se prav tako kreira ob prvem zagonu. Varnostne skupine so specifična konfiguracija na EC2 in predstavljajo požarni zid na nivoju oblaka. Varnostna skupina, ki jo definiramo, mora imeti odprta vsa vrata za IP naslov strežnika, na katerem teče naš sistem, zato da lahko iz sistema dostopamo do virtualk.

Sedaj znamo zgraditi predlogo nove virtualke in jo lahko zaženemo. Za to poskrbi metoda createVm(), prikazana na izpisu kode 4.8.

```
@TransactionTimeout(3600)
@Override
public Vm createVm(VmSetting vmSettings) {

  TemplateBuilder templateBuilder;
  try {
        templateBuilder = prepareTemplate(vmSettings);

        Template template = templateBuilder.build();
        NodeMetadata node = getOnlyElement(compute
          .createNodesInGroup(vmSettings.getGroupName(), 1,
              template));
        logger.info("New node " + node.getId() + " " +
            concat(node
          .getPrivateAddresses(),
              node.getPublicAddresses())
          +" in group "+vmSettings.getGroupName());
```

```
        uploadFile(node, "bootstrap_" +
            vmSettings.getAmiId() + ".sh", true, false);
        uploadFile(node, "init_" + vmSettings.getAmiId() +
            ".sh", true, false);
        uploadFile(node, "loadmonitor_" +
            vmSettings.getAmiId() + ".sh", true, false);
        uploadFile(node, "loadmonitor.zip", false, true);
        uploadFile(node, "startmonitor.sh", true, false);
        uploadContainerFile(node,
            vmSettings.getContainerFile());

        executeBootstrapScript(node, vmSettings);
        executeInitScript(node, vmSettings);
        executeLoadScript(node, vmSettings);

        logger.info("Node in group "+node.getGroup()+"
            initiallized.");

        TypedQuery<build.cross.models.jpa.KeyPair> query =
          em.createNamedQuery("KeyPair.findByName",
              build.cross.models.jpa.KeyPair.class);
          query.setParameter("name", "crossbuild");
          build.cross.models.jpa.KeyPair kp =
              query.getResultList().get(0);

    Vm vm = new Vm();
    vm.setProvider(CLOUD_PROVIDER);
    vm.setCloudId(node.getId());
    vm.setGroupName(vmSettings.getGroupName());
    vm.setIp(node.getPublicAddresses().iterator().next());
    vm.setKeyPair(kp);
    vm.setVmSetting(vmSettings);
```

```
            em.persist(vm);

            return vm;
        } catch (ServiceException | IOException |
            RunNodesException e) {
            logger.severe("Error creating new vm");
            e.printStackTrace();
        }
        return null;
    }
```

Izpis kode 4.8: Metoda createVm() razreda VmManagerSb.

Najprej pridobimo predlogo virtualke s pomočjo metode prepareTemplate(). Nato s klicem createNodesInGroup() na instanci ComputeService sprožimo najemanje nove virtualke. Kot parametre podamo ime skupine, število virtualk in predlogo. Ime skupine je definirano v že omenjenem razredu VmSettings in je enako "crossbuild-amiId". Vsaka virtualka sistema se torej kreira pod imenom skupine s predpono crossbuild, nato pa se doda še identifikator platforme. Z imenom skupine bomo lahko kasneje pridobivali podatke o vseh virtualkah ene in druge platforme in nad imeni skupin računali povprečja zasedenosti. Predpona je pomembna zato, da so virtualke ločene od že obstoječih, ki jih na našem AWS računu morda že uporabljamo za kak drug namen. Ko se klic konča sledi nalaganje vseh potrebnih datotek na novo virtualko. Te datoteke so:

- bootstrap_amiId.sh je skripta, ki namesti vso potrebno programsko opremo,
- init_amiId.sh je skripta, ki zažene našo programsko opremo in po potrebi opravi še nekatere spremembe konfiguracije,
- loadminitor.zip je program za monitoring zasedenosti (njegovo delovanje si bomo ogledali v poglavju 4.6.3),
- startmonitor.sh je skripta, ki zažene komponento za monitoring.

Metodi za nalaganje datotek podamo instanco virtualke, ime datoteke in boolean spremenljivko, ki pove, ali naj se pred nalaganjem zamenjajo DOS znaki za novo vrstico z UNIX znaki. To je potrebno zato, ker smo sistem razvijali na platformi Windows in je

pri zaganjanju skript na virtualkah prihajalo zaradi napačnih znakov za konec vrstice do problemov. Metodi tudi povemo, ali gre za binarno datoteko ali ne, saj je branje iz internih virov glede na tip datoteke implementirano različno. Po uspešnem nalaganju skript se te na virtualki zaženejo, na koncu pa kreiramo še entiteto tipa Vm, ki se shrani v bazo v tabelo vms z vsemi potrebnimi podatki. V nasprotnem primeru bi morali podatke o najetih virtualkah pridobivati direktno od oblačnega ponudnika, kar se izvede kot precej časovno potraten REST klic.

```
#!/bin/sh
# FreeBSD 10
#install packages
pkg install -y sudo
pkg install -y bash
pkg install -y openjdk8-jre
pkg install -y unzip
pkg install -y screen
pkg install -y ezjail

#setup sudo and bash
ln /usr/local/bin/bash /bin/bash
echo "ec2-user ALL=(ALL) NOPASSWD:ALL" >>
   /usr/local/etc/sudoers
```

Izpis kode 4.9: Bootstrap skripta za FreeBSD.

```
#!/bin/bash
# UBUNTU 14.04
#install packages
add-apt-repository -y ppa:openjdk-r/ppa
apt-get update
apt-get -y install openjdk-8-jdk
apt-get -y install unzip
apt-key adv --keyserver
   hkp://p80.pool.sks-keyservers.net:80 --recv-keys
   58118E89F3A912897C070ADBF76221572C52609D
truncate --size=0 /etc/apt/sources.list.d/docker.list
echo 'deb https://apt.dockerproject.org/repo ubuntu-trusty
   main' | sudo tee --append
   /etc/apt/sources.list.d/docker.list
apt-get -y update
apt-get -y install docker-engine
service docker start
```

Izpis kode 4.10: Bootstrap skripta za Ubuntu.

Vsebina skript je predstavljena na izsekih kode 4.9 in 4.10. Obe bootstrap skripti namestita Javo 8 za delovanje monitoring komponente, unzip za dekompresijo monitoring komponente ter še nekatera dodatna orodja za lažjo administracijo. Ubuntu skripta poleg tega namesti še Docker za upravljanje z Docker kontejnerji, FreeBSD pa ezjail za upravljanje z zapori. Skripta za FreeBSD mora namestiti tudi sudo in bash, saj privzeto nista nameščena. To povzroča preglavice knjižnici JClouds, ki pričakuje, da sta obe orodji že nameščeni. JClouds privzeto skripte izvede z lupino bash in pričakuje, da je uporabnik, ki skripto izvaja, v sistemski uporabniški skupini sudoers (angl. *super users*, uporabniki z root pravicami). Ker na sistemu FreeBSD uporabnik "ec2-user" še ni dodan v to skupino, to storimo v zadnji vrstici skripte. Prav zaradi teh težav smo zagonske skripte ločili na boostrap in init del. Po zagonu skripte bootstap na sistemu FreeBSD se začne JClouds obnašati enako kot na Linux sistemih.

Init skripta odpakira komponento za monitoring in jo skupaj s skripto za zagon komponente premakne v mapo loadmonitor. Komponento zažene v novem terminalu z orodjem screen. Če tega ne storimo, bi se program zaprl ob zaključku SSH seje. Nazadnje ukaz za zagon dodamo še v storitev cron, ki bo program samodejno zagnal ob morebitnem ponovnem zagonu virtualke. Primer ukazov za Ubuntu najdemo na izpisu kode 4.11, za FreeBSD pa je skripta skoraj identična.

```
#!/bin/bash
# UBUNTU 14.04
#run loadmonitor
unzip loadmonitor.zip
chmod +x startmonitor.sh
mv startmonitor.sh ./loadmonitor
cd loadmonitor && screen -dmS loadmonitor ./startmonitor.sh

#loadmonitor restart
echo "@reboot cd /home/ubuntu/loadmonitor/ &&
   /usr/bin/screen -dmS loadmonitor ./startmonitor.sh" >>
   loadcron
crontab loadcron
```

Izpis kode 4.11: Init skripta za Ubuntu.

Po izvedbi vseh ukazov imamo na voljo novo virtualko z že nameščeno vso potrebno programsko opremo in z delujočo komponento za monitoring, katero lahko sedaj kličemo iz našega sistema.

4.6.2 Inicializacija oblaka ob prvem zagonu

Ob prvem zagonu sistema želimo, da se vzpostavijo začetni sužnji za vsako platformo, zato smo ustvarili razred VmInitializer, ki ga v celoti vidimo v izpisu kode 4.12. Razred je anotiran z @Singleton in @Startup kar pomeni, da bo to na aplikacijskem strežniku edina instanca razreda in da se bo instanciral ob zagonu aplikacijskega strežnika. Po instanciranju razreda se sproži metoda init() anotirana s @PostConstruct, ki zažene časovnik in sproži metodo firstRun(). Izvedbo metode smo zakasnili zato, da smo prepričani, da se

je aplikacijski strežnik dokončno postavil. V metodi firstRun() najprej preverimo, če že obstajajo vnosi v bazi, sicer kreiramo vnose za podprte platforme, SSH ključe ter varnostne skupine pri oblačnem ponudniku. Generirane SSH ključe dodamo še v Jenkins, ki jih bo potreboval za povezovanje na kontejnerje. Sledi najemanje ene virtualke za vsako podprto platformo. Za hitrejše delovanje uporabimo metodo parallelStream(), ki sproži najemanje v ločenih nitih. Na koncu še odstranimo časovnik, zato da se metoda ne sproži več.

```
package build.cross.services.cloud.management;

@Singleton
@Startup
public class VmInitializer {

  @EJB
  private VmManagerSbLocal vmm;

  @EJB
  private KeypairsAndSgInitLocal kpsg;

  @EJB
  private SecurityMediationSbLocal jenkinsSecurity;

  private Logger logger =
     Logger.getLogger(VmInitializer.class.getSimpleName());

  @Resource
  private TimerService timerService;

  @PostConstruct
  private void init() {
        timerService.createTimer(15000, 600000);
  }
```

```
  @PersistenceContext(unitName = "crossbuild")
  private EntityManager em;


  @Timeout
  @TransactionTimeout(1500)
  private void firstRun(Timer timer) {

  List<VmSetting> settings = em.createNamedQuery(
    "VmSetting.findAll",    VmSetting.class).getResultList();

  if (settings.size() == 0) {
        settings = kpsg.createInitialVmSettings();
        kpsg.createKeypairs();
        kpsg.createSecurityGroups();

        jenkinsSecurity.addGlobalCredentials();

        settings.parallelStream().forEach(s ->
           vmm.createVm(s));
  }
  timer.cancel();
}
```

Izpis kode 4.12: Razred VmInitializer.

4.6.3 Monitoring

Komponento za monitoring namestimo na vsako novo virtualko. V repozitoriju kode jo najdemo v mapi *loadmonitor*. Izpostavlja REST vmesnik, katerega kličemo iz glavnega sistema za pridobivanje podatkov o zasedenosti virtualke. Komponenta je razvita s pomočjo ogrodja KumuluzEE v obliki mikrostoritve, kar v praksi pomeni, da za njeno postavitev ne potrebujemo aplikacijskega strežnika in jo zaženemo kot jar arhiv. S tem

prihranimo nekaj 100MB prostora na vsaki virtualki. Komponenta izpostavlja naslednje REST vire:

- /v1/load/cpu,
- /v1/load/memory,
- /v1/load/disk.

Če želimo izvedeti trenutno zasedenost diska na neki virtualki torej kličemo npr. *http://ip.virtualke/v1/load/disk*. Vse tri metode vračajo procentualno zasedenost v obliki decimalnega števila. Na izpisu kode 4.13 je prikazana implementacija vseh treh metod. Ta je zelo preprosta, saj lahko zasedenost CPU in pomnilnika pridobimo iz javanskega razreda OperatingSystemMXBean, zasedenost diska pa iz razreda File.

```
@Path("/cpu")
@GET
public Response cpuLoad() throws IOException {

 MBeanServerConnection mbsc =
    ManagementFactory.getPlatformMBeanServer();
 OperatingSystemMXBean os =
    ManagementFactory.newPlatformMXBeanProxy(
  mbsc, ManagementFactory.OPERATING_SYSTEM_MXBEAN_NAME,
     OperatingSystemMXBean.class);

 double cpuLoad = os.getSystemCpuLoad();
 return Response.ok(cpuLoad).build();
}

@SuppressWarnings("restriction")
@Path("/memory")
@GET
public Response memoryLoad() throws IOException {
```

```
 MBeanServerConnection mbsc =
    ManagementFactory.getPlatformMBeanServer();
 OperatingSystemMXBean os =
    ManagementFactory.newPlatformMXBeanProxy(
  mbsc, ManagementFactory.OPERATING_SYSTEM_MXBEAN_NAME,
     OperatingSystemMXBean.class);

 double memLoad = (double)(os.getTotalPhysicalMemorySize()
   - os.getFreePhysicalMemorySize()) /
      os.getTotalPhysicalMemorySize();

 return Response.ok(memLoad).build();
}

@SuppressWarnings("restriction")
@Path("/disk")
@GET
public Response diskLoad() throws IOException {

 File f = new File("/");
 double diskLoad = (double)(f.getTotalSpace() -
    f.getUsableSpace()) / f.getTotalSpace();

 return Response.ok(diskLoad).build();

}
```

Izpis kode 4.13: REST metode komponente za monitoring.

Na strani upravljalca oblaka implementiramo še razred LoadTimerSb, ki ga najdemo v paketu *build.cross.services.cloud.metircs*. Razred ima dve metodi, updateLoadData(), ki od vseh virtualk pridobi trenutno zasedenost in jo shrani v bazo ter evaluateLoad(), ki izračuna zgodovinsko povprečje za vsako platformo. Ko povprečna zasedenost kateregakoli vira ene platforme preseže 80%, se sproži najemanje nove virtualke. Metodi se

prožita periodično na nekaj minut s pomočjo anotacije @Schedule.

4.6.4 Upravljanje s kontejnerji

Za upravljanje z Docker in Jail kontejnerji skrbi javansko zrno ContainerManagementSB iz paketa *build.cross.services.cloud.containers*. Vsebuje metodi za izbris in dodajanje novih kontejnerjev. Metodo za dodajanje novih kontejnerjev si poglejmo v izpisu kode 4.14. Zaradi obsežnosti metode smo komentarje o delovanju za lažje sledenje kodi dodali kar v sam izpis.

```
@Override
public Project createNewContainer(Project project) throws
    ServiceException {

/* 1. Preveri če izbrana platforma za projekt sploh
    obstaja. ID se pošlje s strani klienta in bi zato lahko
    bil napačen.
*/
  VmSetting vmSetting = em.find(VmSetting.class,
      project.getVmSetting().getId());
  if (vmSetting==null) {
        throw new ServiceException("Invalid vm settings
            id.");
  }

/*
  2. Zaženi SQL poizvedbo, ki najde najmanj zasedeno
     virtualko. V primeru da ta ne obstaja, se sproži
     izjema tipa ServiceException. Ta se propagira vse do
     spletnega vmesnika, kjer se izpiše napaka. SQL
     poizvedba povpreči zasedenost vseh treh virov (CPU,
     RAM, disk) in rezultate uredi po naraščajoči
     vrednosti. V zanki se še enkrat preveri, ali ima
     virtualka na voljo dovolj diska, ker je to edini vir,
     ki bi povzročil napačno delovanje ob preveliki
```

```
     zasedenosti.
*/
  Query query =
     em.createNamedQuery("LoadHistory.findMinLoad");
  query.setParameter("groupName", vmSetting.getGroupName());

  List<Object[]> lhList = query.getResultList();
  LoadHistory lh = null;
  if (lhList.size()==0) {
        throw new ServiceException("No slaves available");
  } else {
        for (Object[] l : lhList) {
          if ((Double)l[1] <
              EvaluateLoadTimerSb.DISK_TRESHOLD) {
                lh = em.find(LoadHistory.class, l[0]);
                break;
      }
        }
  }
  if (lh==null) {
        throw new ServiceException("No slaves available");
  }

  NodeMetadata node = vmm.getNodeFromVm(lh.getVm());

/*
3. Poimenujemo kontejner. Ime je rezultat zgoščevalne
   funkcije, v katero damo ime uporabnika, ime projektne
   skupine in ime platforme. Ime mora biti unikatno, zato
   se pri dodajanju istega projekta, z istim imenom, na
   isto platformo, zgodi napaka. Nazadnje se pred ime doda š
   e črka c (c kot container), ker se ime zapora na
   platformi FreeBSD ne sme začeti s številko, kar se lahko
```

```
    zgodi pri kodiranju imena s šestnajstiškimi znaki. Tako
    zgenerirano ime se v ukaz za kreiranje kontejnerja
    vstavi namesto oznake %NAME%.
*/
  String cmd = vmSetting.getCreateContainerCmd();
  // %NAME% %IP% %PORT%
  String containerName = project.getProjectGroup().getUser()
    .getId() +project.getProjectGroup().getName()
    +vmSetting.getGroupName();
  MessageDigest md;
  try {
        md = MessageDigest.getInstance("MD5");
  } catch (NoSuchAlgorithmException e1) {
        throw new ServiceException("MD5 alghoritm not
            found");
  }
  byte[] digest = md.digest(containerName.getBytes());

  containerName = Hex.encodeHexString(digest);
  containerName = "c"+containerName; //name cannot be
     numeric for jail
  cmd = cmd.replace("%NAME%", containerName);

/*
4. Generiranje IPja in TCP vrat. Poiščemo nazadnje
    ustvarjen kontejner na izbrani virtualki in njegov IP
    povečamo za 1. Enako naredimo s TCP vrati. Zaradi
    možnosti izbrisa kontejnerjev bi bilo potrebno
    pridobivanje nezasedenih IPjev in vrat izboljšati, saj
    bi lahko prišlo do slabega izkoristka, ko dosežemo š
    tevilko vrat 65535, kljub temu da obstajajo še
    nezasedena vrata, ki so pripadala izbrisanim
    kontejnerjem. Dobljen IP naslov in vrata vstavimo v ukaz
```

```
   za ustvarjanje kontejnerja namesto oznak %IP% in %PORT%.
*/
  TypedQuery<Container> queryC =
     em.createNamedQuery("Container.findLatest",
     Container.class);
  queryC.setMaxResults(1);
  queryC.setParameter("vmId", lh.getVm().getId());
  List<Container> containers = queryC.getResultList();

  InetAddress nextIp = null;
  try {
        nextIp = InetAddress.getByName("192.168.0.1");
  } catch (UnknownHostException e) {
        throw new ServiceException("Could not parse initial
           IP");
  }
  Integer port = 10000;

  if (containers.size()>0) {
        try {
      Container latestContainer = containers.get(0);
      InetAddress lastIp =
         InetAddress.getByName(latestContainer.getIp());
      nextIp = InetAddresses.increment(lastIp);
          port = latestContainer.getPort()+1;
    }
        catch (IllegalArgumentException |
           UnknownHostException ex) {
          throw new ServiceException("Error creating new
             container, reason: invalid IP");
        }
  }
```

```
  cmd = cmd.replace("%IP%",
     InetAddresses.toAddrString(nextIp));

  if (port>65535) {
        throw new ServiceException("Error creating new
           container, reason: invalid port");
  }
  cmd = cmd.replace("%PORT%", port.toString());

  TypedQuery<Container> queryCF =
     em.createNamedQuery("Container.findByName",
     Container.class);
  queryCF.setParameter("name", containerName);
  if (queryCF.getResultList().size()>0) {
        throw new ServiceException("Project with this name
           already exists.");
  }

/*
5. Dobljen ukaz izvedemo na virtualki. Ukaz za ustvarjanje
   kontejnerja je specifičen za vsako platformo posebej in
   je shranjen v bazi v tabeli vm_settings. V shranjenem
   ukazu smo zamenjali besede %NAME%, %IP% in %PORT% s
   pravilnimi podatki.
Primer ukaza za Docker je:

docker run -d -p %PORT%:22 --name %NAME% base

Primer ukaza za FreeBSD Jail je nekoliko bolj zapleten:

ezjail-admin create -f base %NAME% 'lo1|%IP%' &&
   ezjail-admin start %NAME%
```

```
Kot vidimo Docker ne potrebuje podatka o IPju, saj ga
    določi sam. IP nas v resnici niti ne zanima, saj do
    kontejnerjev dostopamo prek javnega IPja virtualke na
    določenih vratih. Prost IP torej iščemo le za
    kontejnerje na platformi FreeBSD.
Na platformi FreeBSD moramo naknadno poskrbeti še za
    pravilno usmerjanje na vrata in sicer s pomočjo
    požarnega zidu pf, na primer:

rdr pass on xn0 proto tcp from any to $IP_PUB port %PORT% ->
%IP% port 22' >> /etc/pf.conf && pfctl -f /etc/pf.conf

Ukaz preusmeri zahteve iz javnega IPja virtualke na nekih
    vratih na notranji IP kontejnerja na vrata 22 (SSH).
*/
  logger.info("Creating new container with command: "+cmd);
  Integer exitStatus = vmm.executeScriptAsRoot(node,
     vmSetting, cmd);

  if (exitStatus!=0) {
        throw new ServiceException("Error creating a
           container.");
  }

  /*
  6. Kontejner shranimo v bazo.
  */
  TypedQuery<KeyPair> queryK =
     em.createNamedQuery("KeyPair.findByName",
     KeyPair.class);
  queryK.setParameter("name", "crossbuild_container");
  KeyPair kp = queryK.getResultList().get(0);
```

```
  Container c = new Container();
  c.setIp(InetAddresses.toAddrString(nextIp));
  c.setName(containerName);
  c.setPort(port);
  c.setProject(project);
  c.setVm(lh.getVm());
  c.setKeyPair(kp);

  project.setContainer(c);

  return project;
}
```

Izpis kode 4.14: Metoda za kreiranje novega kontejnerja.

5 Zaključek

5.1 Končne ugotovitve

V diplomskem delu smo analizirali, zasnovali in razvili sistem za sprotno integracijo izvorne kode na več platformah. Najprej smo v uvodu ugotavljali, zakaj je sprotna integracija uporabna in kakšni so njeni pozitivni učinki pri medplatformskih projektih. Za večjo uporabnost rešitve smo si zadali ključne cilje kot so javna dostopnost, skalabilnost in izolacija projektov. S pomočjo diagramov smo teoretično zasnovali potek delovanja in zaledno arhitekturo sistema. Okvirno smo zastavili tudi izgled spletnega vmesnika. Izvajanje sprotne integracije smo izvedli z uporabo orodja Jenkins, ki je industrijski standard na tem področju. Javno dostopnost sistema smo zagotovili z razvojem spletnega vmesnika, GitHub prijavo in s postavitvijo demo strani. Problem skalabilnosti smo rešili z monitoringom zasedenosti in avtomatičnim najemanjem virtualk v oblaku. Izolacija projektov je bila dosežena s pomočjo kontejnerskih tehnologij Docker in Jails, saj se vsak projekt gradi in testira v lastnem kontejnerju in je tako popolnoma izoliran od ostalih. Podpora operacijskim sistemom iz dveh različnih družin (Linux, Unix) in uporaba dveh različnih kontejnerskih tehnologij priča o razširljivost sistema z dodatnimi platformami

v prihodnje.

Rezultat diplomskega dela je na voljo kot odprtokoden projekt skupaj z demo stranjo, ki je dostopna na naslovu `http://cross.build`. Sistem je že v praktični uporabi za peščico odprtokodnih projektov, h katerim v prostem času prispeva avtor diplome (npr. PvPGN, strežnik za igranje iger in StormLib, kompresijska knjižnica). Preizkusi ga lahko vsak, potrebuje pa le GitHub račun za prijavo. Zaradi odprtokodne narave projekta si lahko vsakdo sistem namesti na lastne strežnike s pomočjo Docker skript, ki so del repozitorija.

5.2 Smernice za prihodnost

Med razvojem smo zaradi časovnih omejitev in obsega dela poenostavili določene dele sistema ali izpustili naprednejše funkcionalnosti, ki niso bile nujno potrebne za delovanje. Možnosti za izboljšave in razširitve je zato precej. Nekatere izmed njih so:

- podpora za platformo Windows Server 2016, ko bo uradno izdana,
- pametnejše zaznavanje zasedenosti virtualk in najemanje novih,
- uporabnikom izpostaviti naprednejše funkcionalnosti, ki jih ponuja Jenkins,
- nadomestiti izpraševanje git repozitorijev z *git post-commit hook-om* (po vsakem *commit-u* se izvede še dodatna skripta, ki bi lahko klicala naš REST vmesnik in s tem sprožila grajenje), saj je izpraševanje počasno in neskalabilno,
- dovoliti uporabniku, da ob dodajanju projekta specificira git branch, ki ga želi graditi,
- poiskati način, kako iz Jenkinsa pridobiti seznam nazadnje grajenih projektov za prikaz na statusi strani,
- podpora ARM platformam in operacijskim sistemom, ki jih ne najdemo pri oblačnih ponudnikih,
- uporaba IPv6 za kontejnerje namesto IPv4+NAT,
- dokumentacija namestitve in uporabe REST vmesnikov.

Veliko je še odprtih vprašanj glede obravnavanja zasedenosti posameznih virtualk. Kljub temu, da na zasedene virtualke ne nameščamo novih projektov, lahko obstoječi projekti organsko prerastejo zmogljivosti virtualke. Trenutno ni mehanizma, ki bi znal prerazporejati obstoječe kontejnerje na manj zasedene instance. Maksimalna zasedenost, ki sproži najemanje in število novih virtualk sta trenutno fiksni števili, verjetno pa bi morali biti dinamični in se prilagajati trendom uporabe sistema.

Nerešeno je tudi vprašanje varnosti proti zlonamernim uporabnikom. Nad kontejnerji se trenutno ne izvajajo nobeni dodatni varnostni ukrepi, prav tako se ne omejujejo viri in uporabnik lahko izvaja poljubne ukaze. Prav tako se ne omejuje število projektov, ki jih lahko uporabnik doda. Administracijska orodja za blokiranje uporabnikov in odstranjevanje zlonamernih projektov niso bila razvita. Pred resno produkcijsko uporabo bi bilo treba nujno nasloviti te varnostne probleme.

Literatura

[1] "10 hosted continuous integration services for a private repository." http://www.yegor256.com/2014/10/05/ten-hosted-continuous-integration-services.html. Accessed: 2015-12-22.

[2] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[3] "Continous integration." http://guide.agilealliance.org/guide/ci.html. Accessed: 2015-12-22.

[4] A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE '08. Conference*, pp. 289–293, Aug 2008.

[5] S. Janiyan, "Adventures in building open source software." https://www.youtube.com/watch?v=-HMXyzybgdM. Accessed: 2016-05-13.

[6] J. F. Smart, *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

[7] A. Mouat, *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, 2016.

[8] "Docker." https://wiki.freebsd.org/Docker. Accessed: 2015-12-22.

[9] "Usage share of operating systems." https://en.wikipedia.org/wiki/Usage_share_of_operating_systems. Accessed: 2016-02-27.

[10] "Windows containers quick start - docker." https://msdn.microsoft.com/en-us/virtualization/windowscontainers/quick_start/manage_docker. Accessed: 2016-02-27.